Towards the Quantum Machine: Using Scalable Machine Learning Methods to Predict Photovoltaic Efficacy of Organic Molecules

Towards the Quantum Machine: Using Scalable Machine Learning Methods to Predict Photovoltaic Efficacy of Organic Molecules

A thesis presented by

Michael Tingley

to
the Department of Computer Science of Harvard University
in partial fulfillment of the honors requirements
for the joint degree of
Bachelor of Arts
in Computer Science and Statistics

Harvard College
Cambridge, Massachusetts

April 1, 2014

Abstract

Recent advances in machine learning have resulted in an upsurge of interest in developing a "quantum machine", a technique of simulating and predicting quantum-chemical properties on the molecular level. This paper explores the development of a large-scale quantum machine in the context of accurately and rapidly classifying molecules to determine photovoltaic efficacy through machine learning. Specifically, this paper proposes several novel representations of molecules that are amenable to learning, in addition to extending and improving existing representations. This paper also proposes and implements extensions to scalable distributed learning algorithms, in order to perform large scale molecular regression. This paper leverages Harvard's Odyssey supercomputer in order to train various kinds of predictive algorithms over millions of molecules, and assesses cross-validated test performance of these models for predicting photovoltaic efficacy. The study suggests combinations of representations and learning models that may be most desirable in constructing a large-scale system designed to classify molecules by photovoltaic efficacy.

Acknowledgments

Big thanks to my thesis advisor Ryan Adams, who helped me find such an interesting opportunity at the intersection of systems and statistical machine learning. He is doing some incredible work in the way that we view and understand molecules, and I'm really looking forward to what will come out of his lab in the near future. Also a huge thanks to Margo Seltzer, my academic advisor and thesis reader, for advice on the thesis process in general and for spending so much of her personal time to offer guidance and discussion. Also great thanks to Luke Bornn for providing a statistical perspective on machine learning.

I also want to extend thanks to my family and friends for keeping me sane during this process, and of course for helping to proofread and critique my thesis. I definitely would not have been able to do this without you all.

Contents

List of figures

List of tables

1 Introduction

As both the global population size and energy use per individual rise, increasing scientific effort is being applied to find cheaper and more sustainable sources of energy. One promising approach is solar photovoltaics, which is the method of converting the energy inherent in the sun's rays into usable electrical energy [1]. A major limitation of solar photovoltaics, however, is the relative inefficiency at extracting electrical energy from incident photons. In response, this paper seeks to utilize scalable machine learning systems in order to rapidly identify molecules that may offer promising characteristics as primary agents in solar photovoltaics.

Photovoltaics exploit the photovoltaic effect in order to output electrical energy. The photovoltaic effect is caused by incident photons dislodging electrons from the photovoltaic material. These moving photons can then be used to create a potential difference and induce a current. The material from which electrons are dislodged is referred to as the primary agent. Theoretically, any compound can serve as a primary agent for strong enough intensities of incident light; however, in most compounds, the photovoltaic effect is too inefficient to be used effectively [2]. Choosing an efficient primary agent in solar cells is a difficult task due to the large spectrum of possible candidate compounds. The efficacy of a photovoltaic material is frequently measured in *percent conversion efficiency* (PCE), the fraction of potential energy in a photon incident to the solar cell that can be converted into electric energy [1]. Photovoltaics have primarily been manufactured using silicon and other semimetals, due to their relatively high PCE and material lifetimes. Crystalline silicone is able to achieve more than 15% PCE with a lifetime of over 25 years [3]. Comparatively, the 2013 Solar Cell Efficiency Tables report average PCE for organic-based solar cells to be around 5% [4]. This value is too low for widespread use in most areas of interest for photovoltaics. However, carbon-based photovoltaics provide a spectrum of advantages, including inexpensiveness and ease of manufacturing. Further, the material is

flexible, and can therefore be applied in ways that rigid silicon-based photovoltaics cannot. Additionally, the manufacturing process for silicon-based photovoltaics requires highly specialized production machinery [5].

The major limiting factor in developing carbon-based photovoltaics is that there is not a simple procedure for evaluating the effectiveness of a candidate material to be used as the primary agent. Selection of an organic photovoltaic candidate is "predominantly based on empirical intuition, professional inspiration, or experience with certain compound families" [6]. Thus, in order to assess the performance of a photovoltaic candidate, a solar cell must be engineered using that candidate as the primary agent, and then tests must be manually undertaken to evaluate performance. This entire procedure is a multi-month process [6].

The Harvard Clean Energy Project has developed a platform for performing distributed simulations of this process using the IBM World Community Grid. This works by replicating a molecule *in silico*, and modeling first-principles interactions between atoms using 150 million density functional theory calculations. However, this simulation-based process can still take a month or longer per molecule to achieve reliable PCE numbers [6]. These times are too long to make reasonable headway into this problem, especially considering that there is a domain of many millions of candidate organic molecules.

Recent work in machine learning of quantum-molecular features offers hope for reliably predicting chemical properties without an extensive simulation period. Using a machine learning approach could serve as a first-pass filter in order to identify promising candidates for further, more rigorous investigation. These endeavors are part of the broader goal of developing a quantum machine, a system capable of efficiently predicting complex chemical characteristics from simple molecular descriptors.

Montavon et al. demonstrate reliable results for learning molecular electronic properties of

chemical compounds, using a number of different intermediate molecular representations and learning approaches [7]. The work by Montavon et al. also exposes some of the particular difficulties of building a quantum machine:

1. The exact target property is often not directly computable even if the relevant covariates are known.
2. Regression learning is not directly possible over standard molecular representations, which are oftentimes strings.
3. Since molecules are highly complex, the amount of data needed for reliable learning may be prohibitively large or require a prohibitively long training period for straightforward learning approaches.

This paper intends to find solutions to these limitations for the specific task of identifying promising photovoltaic candidate molecules. Specifically, this paper proposes several novel representations for a molecule that are amenable to learning and prediction, and extends existing representations proposed in the literature. This paper also proposes implementations of several distributed learning algorithms in order to rapidly perform learning over a large dataset of molecular features. Finally, the most significant contribution of this paper is the development of a scalable statistical machine learning system capable of utilizing a supercomputing cluster to run massively distributed learning experiments over molecular feature space. This paper analyzes the performance of different learning algorithms and molecular representations in order to suggest promising model combinations for performing molecular regression, using the Harvard Clean Energy Project's database of molecular data.

2 Related work

2.1 Molecular representations

Molecules are often represented in compact, string-based formats. This has the advantage that the molecular representations are human-interpretable. A drawback to this string-based representation is that it is challenging to use for machine regression. Much of the related literature is devoted to identifying molecular representations that are amenable to machine learning, and identifying machine learning approaches that offer promising results for molecular regression.

2.1.1 Feature extraction

Ideally, a machine representation of a molecule would succinctly and numerically summarize the molecule. Bergeron et al. demonstrate that, with conscientious variable selection, features that can be rapidly extracted from molecules can serve as an excellent basis for advanced property prediction [8]. This trades off a comprehensive specification of the molecule for a succinct summarization. These basis features can be efficiently computationally extracted using professional cheminformatics toolchains such as ChemAxon [9] and RDKit [10]. The nature of these features is diverse and can include properties such as molecular mass, Merck molecular force field energy, and Van der Waals surface area. Most of these features can be computationally evaluated within several seconds [9].

2.1.2 Coulomb matrix

A downside to the feature extraction approach is that it is somewhat arbitrary and dictated by circumstance — the potential selection of features is limited by the software toolkit being used, and does not exploit the entire structure of a molecule in the machine representation. Rupp et al. have done extensive work in developing a first-principles quantum structure known as the *Coulomb matrix* for representing a molecule in a learnable way [11].

The Coulomb matrix is an electronic structure representation method based on quantum-mechanical first principles. The Coulomb matrix works by modeling charge interactions between atoms in a molecule. It is therefore a relatively simple model that, for a given molecule, requires only the nuclear charges of each atom and their Cartesian coordinates in three dimensional space to be known. Representationally, the Coulomb matrix is a symmetric matrix on which the major diagonal models absolute charge strength of each atom, and off-diagonal entries model the pairwise atomic charge strengths for each pair of atoms [12]. Rupp et al. show that the Coulomb matrix preserves many properties of a good descriptor [11].

Montavon points out that molecules do not necessarily uniquely identify Coulomb matrices; that is, for an individual molecule, there may be multiple possible Coulomb matrix representations [7]. These arise from the fact that molecules may be indexed in an arbitrary order, and therefore the rows and columns of the Coulomb matrix are not unique to each molecule. This is disadvantageous for learning, since it means that a regression algorithm may correctly provide different predictions for two inputs even if they represent the same molecule. Workarounds for this and additional details of this construct are examined in more depth in Section 3.2.2 of this paper.

2.2 Learning approaches

Machine learning has been a topic for learning properties of molecules since the beginning of the 1990s [13]. However, while much research has been put into learning properties of individual molecules or types of molecules, the literature on learning properties across the molecular space is sparse. The main focus of the literature so far has been on using neural networks, linear regression, and Gaussian processes as a basis for machine learning of molecular features.

2.2.1 Neural networks

Artificial neural networks are supervised machine learning algorithms for associating inputs with outputs by a nonlinear model. Neural networks work by iteratively computing differences between predicted and actual values, and refining the underlying nonlinear model based on these differences. Neural networks have shown promising results for molecular regression since at least 2006. Manzhos and Carrington first showed that neural networks could be used to effectively impute molecular potential energy surfaces from a body of quantum-molecular data [14]. Recently, much research has been put into using neural networks to infer electrical-chemical properties from molecules. Since Rupp et al. introduced the Coulomb matrix to model molecular features, neural networks have been extensively used in this space, due to their ability to learn patterns from structured data matrices [11]. Since then, neural networks have been used to learn atomization energies, static polarizabilities, frontier orbital eigenvalues, ionization potentials, electron affinities, and other electronic properties [7, 12, 15].

One main drawback of neural networks is their extensive training time. Even for modestly sized training sets, neural networks can require many training iterations before acceptable predictive power is achieved. It is therefore necessary to employ parallelization approaches in order to tractably perform regression over large datasets. Neural networks have traditionally been parallelized using a method known as "Network Parallel Training" [16]. Network Parallel Training involves replicating each node (or a set of nodes) in the neural network on each processor. Each processor is then responsible for handling the computations associated with that node. This process effectively parallelizes the neural network by dividing up work done by columns of nodes in the network [17]. However, as this process requires extensive inter-process communication, it is not efficient for distributed computation over multiple different machines. Dahl, McAvinney, and Newhall suggest an implementation of "Pattern Parallel Training" for neural networks, whereby the entire

network is duplicated across processes, and each process is responsible for processing a random subset of the input patterns [16]. This is discussed more in Section 3.3.4 of this paper. Furthermore, Niu et al. propose a lock-free gradient descent algorithm known as Hogwild! that could be used, e.g., on each machine within a distributed neural network architecture to speed up backpropagation [18].

Dean et al. and Coates et al. show how to enhance the above approaches with distributed deep learning. In this context, deep learning is an unsupervised learning method capable of learning appropriate initialization weights for neural networks from unlabeled data. Their paper shows how to extend this technique to datasets with millions of inputs by exploiting cluster computation [19, 20]. Previous methods have attempted to leverage distributed computation through GPUs, but have often been limited by the CPU-to-GPU data transfer bottleneck [21]. This class of approaches is useful, because it can utilize, for instance, molecular data whose PCE is unknown in order to improve the initial neural network weights during supervised training.

2.2.2 Linear regression

Linear regression is valuable for its simplicity and interpretability. In learning the chemical feature space, it has shown to be within an order-of-magnitude as effective as other, more sophisticated learning approaches [15]. In addition, it leads to a model with interpretable parameters that does not require significant computational resources to evaluate.

Nonetheless, for large, high-dimensional training sets, computation time for linear regression can be significant, especially when cross-validation and tuning of various regularization parameters is required. This computational cost comes from having to perform large matrix operations over the entire dataset. Most existing implementations of parallelized linear regression are based on exploiting QR-matrix decompositions and performing GPU-distributed concurrent matrix operations [22]. While such GPU-based approaches can

improve the speed of the core algorithm, oftentimes a speed bottleneck comes from the entire training set being too large to fit into RAM. This results in frequent paging in and out of the data from the hard disk, which is computationally costly. Parallel algorithms for linear regression are therefore beneficial not only for speeding up the core algorithm, but also for dividing data across multiple cores such that each smaller dataset fits into the RAM of its respective core. Xu, Miller, and Wegman propose an efficient distributed model for solving linear regression problems by solving subsets of a linear regression problem across different cores [23].

Instituting regularization is important to prevent overfitting. Unregularized linear regression is known as ordinary least squares regression (OLS). Ridge regression is a common and well-known closed-form method of penalizing the ℓ_2-norm of the regression weights. LASSO[1] regression is another form of linear regression, which penalizes the ℓ_1-norm of the weights. In LASSO regression, the regularization penalty is proportional to the sum of the weights, which tends to induce sparsity and pull the weights of the least useful features to zero before affecting the weights of the other features. This is useful for performing regression-based feature selection. Mateos, Bazerque, and Giannakis provide the description for an efficient distributed implementation of ℓ_1-regularized linear regression by having multiple cores iteratively converge to a set of weights, regularized by their magnitudes [24].

2.2.3 Gaussian processes

Gaussian processes are predictive models that maintain a prior over functions to fit the data. Once new data is observed, we can compute the posterior over these functions to determine functions that are the most likely to fit the observed data. Bartók and Payne introduce the use of Gaussian processes to model complex potential energy landscapes for molecules [25]. They demonstrate very promising results using a representation model

[1] LASSO stands for "least absolute shrinkage and selection operator".

achieved by projecting atomic densities onto the surface of a four-dimensional unit sphere. Bartók and Payne discuss that Gaussian processes are of particular interest for their ability to improve faster with more data when compared to other models, in particular neural networks. While Gaussian processes seem to offer promising results in terms of regression, their significant computational complexity makes them difficult to employ in practice, even on modestly-sized data sets. The main cost of this approach comes from having to invert the covariance matrix, which is the size of the data. Gaussian process training incurs cubic time and quadratic memory in the size of the data [26].

Several papers have been devoted to trying to overcome this computational bottleneck, either through approximate serial techniques or parallelization. A standard technique for improving computation time is to use matrix decomposition or matrix-vector multiples to approximately invert the covariance matrix. Murray overviews some techniques that use approximately sparse matrix kernels in computation of Gaussian processes. These techniques generally involve inducing sparsity on the inputs (or approximating them with a sparse representation), partitioning the inputs into smaller, easier to solve Gaussian process problems and then merging them, or choosing particular kernel functions that lead to covariance matrices that can be approximately inverted quickly. However, these approaches suffer from poor scalability to large datasets or poor approximations with high-dimensional inputs [27]. Bo and Sminchisescu demonstrate an efficient serial method of approximately solving Gaussian processes using greedy block coordinate descent. This is a dense solver, which uses iterative methods instead of inverting the full covariance matrix [26]. Gramancy, Niemi, and Weiss demonstrate several approaches for approximate Gaussian process parallelization using clusters and GPUs. These methods generally involve splitting approximate subproblems across several computers or the GPU. While these tests showed reasonable similarity between the exact and approximate methods for small input sizes (under 10000), due to the complexity of the exact algorithm, it was not possible to

Table 1: Terminology

Representation type	Refers to a way of numerically modeling a molecule. In this project, these are features, Coulomb matrix, Coulomb eigenspectrum, binary adjacency matrix, binary adjacency eigenspectrum, n-ary adjacency matrix, and n-ary adjacency eigenspectrum.
Input	The collection of data that fully describes a single molecule within a representation.
Predictor	The scalar value defined for each dimension of an input.
Feature	A predictor for the features representation.
Response variable	The HOMO-LUMO gap for a specific input.
Core	A single computational unit in a distributed environment.

determine how accurate the parallelized algorithm was for larger input sizes [28]. Overall, Chen et al. show that combining matrix decomposition techniques with a parallelized approach may be the most effective way to extend Gaussian process regression to very large datasets. However, such approaches require either assumptions about the data model or use approximations that may result in significantly degraded predictive performance for very large data sets [29].

2.2.4 Other approaches

Many learning approaches other than those discussed above have been presented in the quantum machine literature. In particular, Hansen et al. provide a comprehensive analysis of the performance of different types of learning approaches [15]. These approaches are divided into roughly four different areas-of-interest: basic learning methods, methods with Gaussian kernels, methods with Laplacian kernels, and neural networks.

3 Experimental approach

This section discusses the specific model and regression choices made throughout this project. It also details the specifics of the dataset and the derivations used to parallelize molecular regression.

In order to disambiguate some of the terms used in this section, please refer to Table 1 for

a list of select terms and their usages in this paper.

3.1 Dataset

The Harvard Clean Energy Project (CEP) is an initiative at Harvard University to identify organic molecules with promising photovoltaic properties. Using the IBM World Community Grid, the CEP has computed various molecular characteristics for over 1.8 million molecules. The CEP has generously made available their dataset for this project.

These molecules are listed in canonical SMILES format. The **S**implified **M**olecular-**I**nput **L**ine-**E**ntry **S**ystem is a compact string-based representation for molecules [30]. This is one of the industry standards for molecular representation, and the initial input for our regression system.

Literature shows that the HOMO-LUMO gap, the difference in energy between the **H**ighest **O**ccupied **M**olecular **O**rbital and the **L**owest **U**noccupied **M**olecular **O**rbital, is a good proxy for photovoltaic efficacy of a molecule [31]. These values take a long time to compute through simulation but, fortunately, the dataset provided by the CEP lists these values. The HOMO-LUMO gap is the response variable that we are targeting in this paper for each molecule.

This project uses the CEP's entire molecular database for analysis in this paper. This database is comprised of 1,824,230 primarily carbon-based molecules. Each molecule is comprised of up to 35 individual atoms.

3.2 Data representations

3.2.1 Feature extraction

From a SMILES string, the ChemAxon cheminformatics toolchain can be used to directly and efficiently extract features from a molecule. There are many features available, and only a subset of them are numeric. Figure 1 lists the 77 available real-valued features and

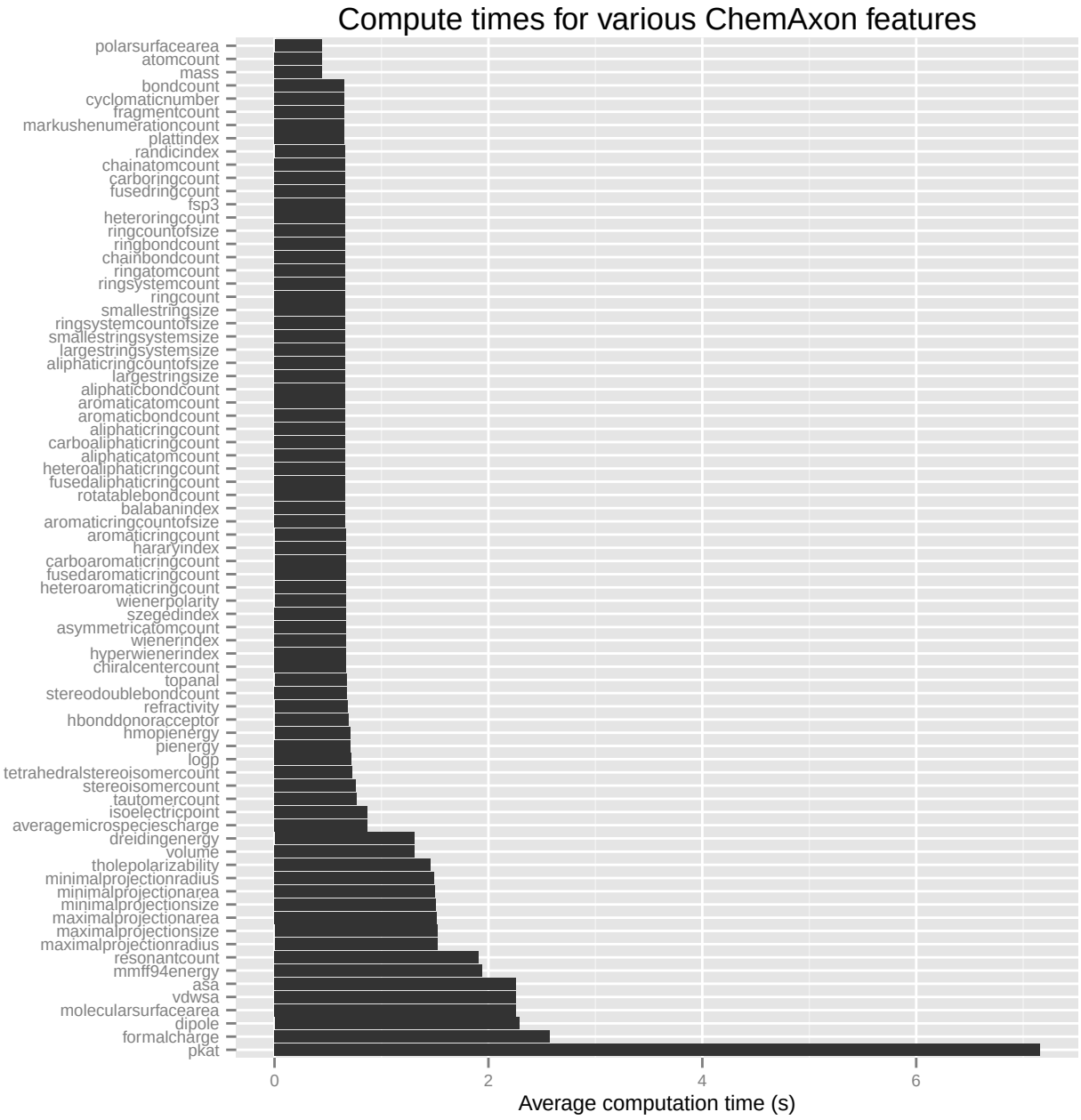


Figure 1: Average per-molecule ChemAxon feature compute times. These are the compute names; the corresponding interpretable names are at `https://www.chemaxon.com/marvin/help/applications/cxcalc-calculations.html`.

their computation times. (Note that these feature names are the compute names; most compute names actually correspond to more than one interpretable molecular feature.) Examining the table, it is clear that extracting features is highly computationally intensive. Even though these features can be computed in parallel, we chose to eliminate a few features based on long computation times. The final list of 59 computed features used in

Table 2: Extracted ChemAxon features. Descriptions can be found at `https://www.chemaxon.com/marvin/help/applications/cxcalc-calculations.html`. Note that these are the interpretable names; the compute names are listed in the above URL.

$a(xx)$	$a(yy)$	$a(zz)$
Acceptor count	Acceptor site count	Aliphatic atom count
Aliphatic bond count	Aliphatic ring count	Aromatic atom count
Aromatic bond count	Aromatic ring count	ASA
ASA_H	ASA_P	ASA+
ASA-	Atom count	Balaban index
Bond count	Chain atom count	Chain bond count
Charge	Dipoles	Donor count
Donor site count	Dreiding energy	Formal charge
FSP3	Harary index	Hetero ring count
Heteroaliphatic ring count	Heteroaromatic ring count	Hyper Wiener index
Largest ring size	log P	Mass
Maximal projection area	Maximal projection radius	Minimal projection area
Minimal projection radius	MMFF94 energy	molecular
pI	Pi energy	Platt index
Polar surface area	Randic index	Refractivity
Ring atom count	Ring bond count	Ring count
Rotatable bond count	Smallest ring size	Szeged index
Van der Waals surface area (3D)	Van der Waals volume	Wiener index
Wiener polarity		

this project are displayed in Table 2.

3.2.2 Coulomb matrix

The Coulomb matrix is a promising representation because it exploits the entire molecular structure in a learning representation for the molecule. The Coulomb matrix can be computed simply from the Cartesian coordinates of the atoms and their atomic charges. For atom number $1 \leq i \leq n$, if we let Z_i be its atomic charge and $\boldsymbol{R}_i$ be its Cartesian coordinates, then, according to [7], we can specify each i, jth element of the $n \times n$ Coulomb matrix as

$$C_{i,j} = \begin{cases} 0.5 Z_i^{2.4} & \text{if } i = j \\ \frac{Z_i Z_j}{|\boldsymbol{R}_i - \boldsymbol{R}_j|} & \text{if } i \neq j \end{cases}$$

There are two major problems with the Coulomb matrix representation. First, since the

matrix is constructed from pairwise interactions of atoms, the matrix for the kth molecule will have dimensions $n_k \times n_k$, where n_k is the number of atoms in the kth molecule. Clearly, the size of the matrix will, in general, be different for different molecules, which makes consistent learning difficult. Hansen et al. propose a simple solution to this problem, by padding all molecules in the dataset with a sufficient number of "dummy atoms" such that all molecules have the same total number of real plus dummy atoms [15]. These dummy atoms have zero independent and pairwise charges.

The second problem is that the Coulomb matrix is not unique — for a single molecule, there exist a number of different, valid Coulomb matrices computable by permuting the atom indices. Montavon et al. propose several Coulomb matrix transformations that circumvent this difficulty [7].

1. **Sorted eigenspectrum.** Although the Coulomb matrix itself depends on the ordering of the atoms, the eigenvalues do not. For each matrix, the eigenvalues can be computed in decreasing order. This is referred to as the *sorted eigenspectrum* representation, and is clearly invariant to the ordering of the atom indices.
2. **Sorted Coulomb matrix.** If we apply a deterministic sorting method to the matrices, then the matrices can be coherently compared. The matrix can be sorted row-wise, where rows with a higher "length" appear earlier. The "length" is computed by considering each row as a vector and computing its ℓ_2-norm.
3. **Random Coulomb matrix.** Finally, Montavon et al. propose utilizing randomly-sorted Coulomb matrices. This is done by randomly permuting the columns and rows of a Coulomb matrix based on the probability of that Coulomb matrix being generated by a random permutation of the atoms.

The sorted eigenspectrum representation has the advantage of being a succinct representation of the matrix, but has the downside of losing some of the information represented in

the Coulomb matrix. Nonetheless, it is amenable to linear regression in addition to neural network regression, and is a simple and robust way of summarizing the Coulomb matrix, and so we choose to use this representation in our analysis. The sorted Coulomb matrix is also fairly straightforward and ensures the comparability of Coulomb matrices, and so we use it in our analysis as well. In order for the random Coulomb matrix representation to produce useful results, the matrix must be replicated many times in the dataset. However, since our dataset is already very large, we chose to exclude this representation for the sake of computational tractability. For smaller datasets, it has been shown that randomly sorted Coulomb matrices can yield an up to tenfold increase in regression performance in certain circumstances, and so this is an area of interest for further research [15].

3.2.3 Adjacency matrix

As examined in the Section 2.1.2, existing representation methods exploit quantum-chemical effects to create the Coulomb matrix. However, this approach ignores important classical chemistry relations of atomic bonds and physical structure of the molecule. In some sense, these features may offer lower fidelity insight into the nature of the molecule; however, this may be viewed as a form of sparsity in the representation type, and is sometimes desirable in machine learning for both improved speed and precision of learning. We propose a new but simple learning representation form for the molecule, the adjacency matrix. This representation is basic and comes in two forms, binary and n-ary.

Binary adjacency matrix. The binary adjacency matrix for a molecule with n atoms is an $n \times n$ symmetric matrix. The diagonals of this matrix are zero, and the i, jth element of this matrix is 1 if there exists a bond of any type between atoms i and j.

n-ary adjacency matrix. The n-ary adjacency matrix is similar to the binary adjacency matrix, except that the i, jth element in the matrix represents the number of bonds between atoms i and j. For instance, if all bonds are single bonds, this is the same as the binary

adjacency matrix. However, if there are double, triple, or aromatic bonds, this matrix will have 2s, 3s, and 1.5s in it respectively.

The resultant adjacency matrices can be transformed in the same way that we transformed Coulomb matrices in order to ensure invariance with respect to the permutation of the atoms.

3.3 Distributed learning architectures

Performing full-data regression on the entire dataset of 1.8 million molecules is very computationally-intensive. Due to time constraints, this study had to be selective on which regressions to perform. All core procedures were parallelized on a massive scale using the Harvard Odyssey cluster (explained more below in Section 3.3.1). Literature suggests that linear regression, neural networks, and Gaussian processes may be among the most promising models for molecular regression. Due to the huge computational cost and difficulty in parallelization associated with Gaussian processes, we decided to use parallelized linear regression techniques and neural networks to predict the HOMO-LUMO gap for molecules.

3.3.1 The Odyssey cluster

This project utilizes the Message Passing Interface (MPI) protocol extensively to perform distributed experiments across multiple machines with communication. In order to exploit the massively parallelized computing capabilities of MPI, this project was deployed on the Odyssey supercomputing cluster, hosted by Harvard University's Faculty of Arts and Sciences Research Computing Team (see [32]). The partition of Odyssey used for this project consists of 28,000 AMD Opteron 6376 "Abu Dhabi" processor cores running on Linux CentOS6. Each computational unit is referred to as a 'core', and may or may not be an individual, physical machine. A node contains 64 cores. Each node is connected via Infiniband high-speed interconnect. Each node has a pool of 256GB of RAM [33]. Jobs are

submitted and queued according to a fairshare-based scheduler running the Simple Linux Utility for Resource Management [34].

3.3.2 Parallelizing OLS and ridge regression

Most existing implementations of parallelized OLS are based on exploiting QR-matrix decomposition and performing GPU-distributed concurrent matrix operations [22]. However, in our case, this is limiting in two ways:

1. If the data is so large that a core cannot sufficiently represent the entire data in RAM, then disk communication costs will significantly inhibit computation time.
2. For systems lacking high-availability GPUs like Odyssey, this is an inconvenient option.

For these reasons and drawing on motivations from the work by Xu, Miller, and Wegman [23], we have derived distributed OLS for an MPI-based interface. This has the advantage that one core is not responsible for performing computations over the full dataset. We could also employ GPU-based matrix multiplication techniques on individual cores to further speed up computation.

The distributed linear regression compute network is modeled in Figure 2. This network model assumes m total cores. In the implementation description below, $\mathbf{1}_\ell$ refers to an ℓ-length column vector of ones. $1 \leq k \leq m$ is used to refer to an arbitrary core. Variables subscripted with (k) refer to the set of the variable local to core k. Our goal is to derive the standard OLS regression weights vector used in linear regression,

$$\hat{\boldsymbol{\beta}}_{\text{OLS}} = \left(\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}}\right)^{-1}\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}}.$$

Phase 1. This is the data distribution phase. The root core reads the n inputs, each of which have p predictors, as the $n \times p$ matrix of inputs $\boldsymbol{X}$, and the n-length column vector of

response variables $\boldsymbol{y}$. Note that the inputs $\boldsymbol{X}$ should be centered and rescaled so that each predictor type has mean 0 and variance 1. This ensures that the regularization levels are interpreted correctly and penalize each of the weights independently of the nominal scale of the predictors. The root splits the data into m approximately-equally-sized matrices and distributes it to the other cores using MPI's SCATTER.

Phase 2. This phase completes all computation that can be done without sharing information. The information computed in this phase summarizes the data for succinct computation by other cores in the next phase. Core k receives n_k inputs. Each core then computes the following quantities over the inputs on that core.

- The p-length column vector of predictor column means, $\overline{\boldsymbol{x}}_{(k)} = \boldsymbol{X}_{(k)}^{\mathsf{T}} \mathbf{1}_{n_k} / n_k$. This is the mean for each type of predictor.
- The $n_k \times p$ centered predictor matrix, $\overline{\boldsymbol{X}}_{(k)} = \boldsymbol{X}_{(k)} - \mathbf{1}_{n_k} \overline{\boldsymbol{x}}_{(k)}^{\mathsf{T}}$. This is the matrix of predictor differences from their means.
- The scalar response mean, $\overline{y}_{(k)} = \boldsymbol{y}_{(k)}^{\mathsf{T}} \mathbf{1}_{n_k} / n_k$.
- The n_k-length centered response vector, $\overline{\boldsymbol{y}}_{(k)} = \boldsymbol{y}_{(k)} - \mathbf{1}_{n_k} \overline{y}_{(k)}$. This is the column

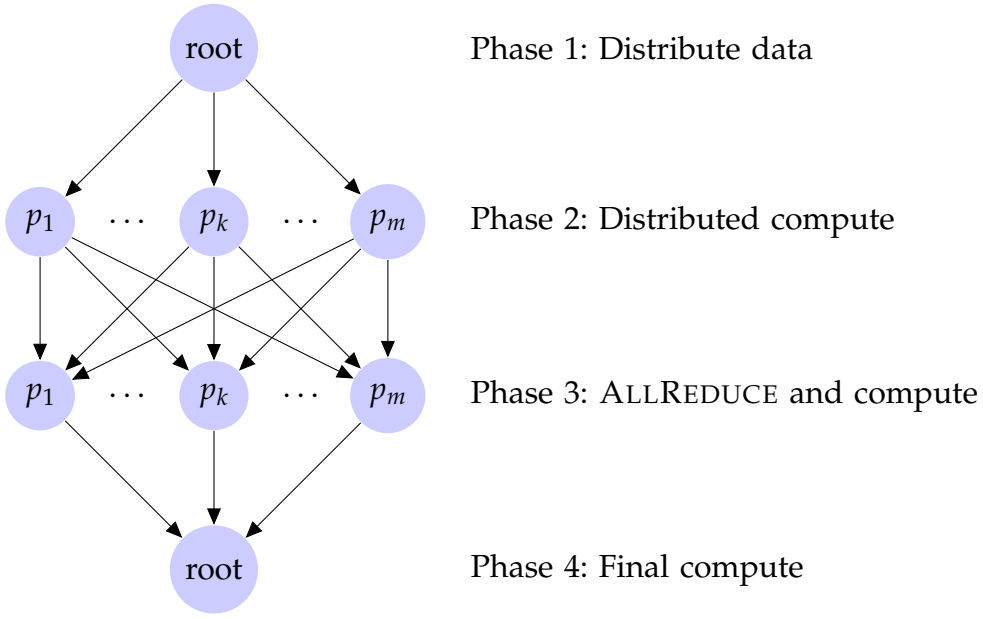


Figure 2: Distributed OLS and ridge regression network architecture

vector of response variable differences from their mean.

- The $p \times p$ square of centered predictors matrix, $\overline{\boldsymbol{X}}_{(k)}^{\mathsf{T}}\overline{\boldsymbol{X}}_{(k)}$. Entry i, j of this matrix is the dot product of the ith and jth centered predictor vectors. So the ith diagonal entry of this matrix is the sum of the squares of the ith centered predictor.
- The p-length total centered predictors times centered responses vector, $\overline{\boldsymbol{X}}_{(k)}^{\mathsf{T}}\overline{\boldsymbol{y}}_{(k)}$. The ith entry of this column vector is the dot product of the ith centered predictor vector and the centered response vector.
- The p-length column vector representing this core's responsibility for each type of predictor, $\text{InResp}_{(k)} = n_k\overline{\boldsymbol{x}}_{(k)}/n$.
- The scalar representing this core's responsibility for the response variable, $\text{OutResp}_{(k)} = n_k\overline{y}_{(k)}/n$.

Phase 3. Using MPI's ALLREDUCE, we efficiently sum up the input and output responsibilities across all cores onto all cores. The sum of the InResp_k's is clearly $\overline{\boldsymbol{x}}$ (the global predictor column means), and the sum of the OutResp_k's is $\overline{y}$ (the global response mean), and so each core now has access to these quantities, in addition to the quantities computed in the previous phase. Each core now computes the following two quantities.

- $n_k(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})(\overline{y}_{(k)} - \overline{y})$
- $n_k(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})^{\mathsf{T}}$

These values can be used in conjunction with those computed in Phase 2 in order to compute the following quantities.

- $\overline{\boldsymbol{X}}_{(k)}^{\mathsf{T}}\overline{\boldsymbol{y}}_{(k)} + n_k(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})(\overline{y}_{(k)} - \overline{y})$. This p-length column vector is the fraction of $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}}$ that can be computed using the data available on the kth core.
- $\overline{\boldsymbol{X}}_{(k)}^{\mathsf{T}}\overline{\boldsymbol{X}}_{(k)} + n_k(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})^{\mathsf{T}}$. This $p \times p$ matrix is the fraction of $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}}$ that can be computed using the data available on the kth core.

Phase 4. We know now that each core owns a fraction of $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}}$ and $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}}$, which are global quantities that we need in order to perform the linear regression. We can compute the value for these quantities that represents the entire dataset simply by summing over the fractions of these values owned by each core.

We use MPI's REDUCE on the root core to sum up remote values across the other cores. Specifically, we have that

$$\begin{aligned}\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}} &= \sum_k \left[\overline{\boldsymbol{X}}_{(k)}^{\mathsf{T}}\overline{\boldsymbol{y}}_{(k)} + n_k(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})(\overline{y}_{(k)} - \overline{y})\right] \\ \overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}} &= \sum_k \left[\overline{\boldsymbol{X}}_{(k)}^{\mathsf{T}}\overline{\boldsymbol{X}}_{(k)} + n_k(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})(\overline{\boldsymbol{x}}_{(k)} - \overline{\boldsymbol{x}})^{\mathsf{T}}\right]\end{aligned}$$

Final compute. Finally, we can compute the maximum likelihood weights for OLS. For regression of the form $y = \beta_0 + \boldsymbol{\beta}_1\boldsymbol{x}$, we have that

$$\begin{aligned}\hat{\boldsymbol{\beta}}_1^{\text{OLS}} &= \left(\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}}\right)^{-1}\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}} \\ \hat{\beta}_0^{\text{OLS}} &= \overline{y} - \overline{\boldsymbol{x}}^{\mathsf{T}}\hat{\boldsymbol{\beta}}_1\end{aligned}$$

Note for the final compute that $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}}$ is a $p \times p$ matrix. This means that the size of the matrix is a function of the number of dimensions in the dataset and not a function of the total number of inputs n. Since p is small, this means that it is computationally easy to perform the required inversion on $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}}$. Furthermore, realize that $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}}$ is a column vector of length p. Therefore, it is computationally easy to multiply $(\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}})^{-1}$ with $\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}}$.

Given this framework, we can easily introduce arbitrary ℓ_2 regularization to perform ridge regression. If $\mathbb{I}_p$ is the $p \times p$ identity matrix, then we can add ℓ_2 regularization of magnitude λ to the linear regression by using

$$\hat{\boldsymbol{\beta}}_1^{\text{ridge}} = \left(\lambda\mathbb{I}_p + \overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{X}}\right)^{-1}\overline{\boldsymbol{X}}^{\mathsf{T}}\overline{\boldsymbol{y}}.$$

(A derivation of this is provided in [35].)

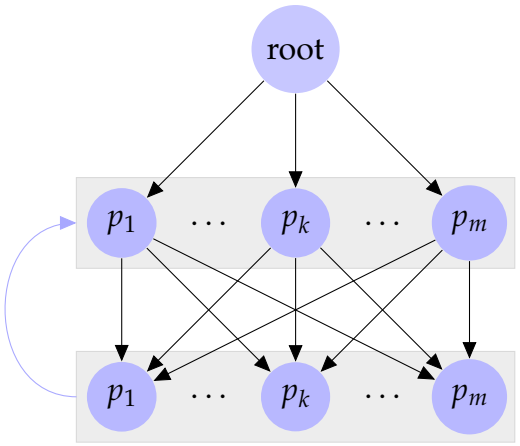
From a Bayesian perspective, Murphy shows that this approach is equivalent to assuming a Gaussian prior on the weights with mean zero and variance σ^2/λ, where σ^2 is the variance of the true weights [35].

3.3.3 Parallelizing LASSO regression

As discussed in Section 2.2.2, LASSO regression is useful for identifying the most important predictors in the dataset. Mateos, Bazerque, and Giannakis provide an efficient distributed implementation for LASSO regression using coordinate descent in [24].

The LASSO regression compute network is modeled in Figure 3. Unlike the other distributed algorithms discussed in the current paper, we use this distributed procedure as described in [24] verbatim. As a result, rather than motivating and deriving the distributed algorithm, we present only the final method here, referring an interested reader to [24] as a reference. The listing is shown in Algorithm 1. This algorithm iteratively refines weight estimates in order to choose the intercept β_0 and weights $\boldsymbol{\beta}$ that fit the LASSO objective for a given regularization level λ, which is defined as

$$\hat{\beta}_0, \hat{\boldsymbol{\beta}} = \arg\min_{\beta_0, \boldsymbol{\beta}} \frac{1}{2} ||\boldsymbol{y} - \mathbf{1}_n \beta_0 - \boldsymbol{X}\boldsymbol{\beta}||_2^2 + \lambda ||\boldsymbol{\beta}||_1,$$



Phase 1: Distribute data, initialize iterates

Phase 2: AllReduce to compute intercept; AllGather to share current weights

Phase 3: Refine weight estimates

Figure 3: Distributed LASSO regression network architecture

Algorithm 1: Distributed Coordinate Descent LASSO

Data: $\lambda > 0$: regularization level
$\boldsymbol{X}$: $n \times p$ matrix of inputs, centered and scaled so each column has mean 0 and variance 1
$\boldsymbol{y}$: n-length vector of responses
m: total number of machines
$c > 0$: penalty coefficient to determine rate of coordinate descent (we use 100 in this paper)

Use MPI to SCATTER $\boldsymbol{X}, \boldsymbol{y}$ to each machine. Machine k receives $\boldsymbol{X}_{(k)}$: $n_k \times p$ matrix subset of $\boldsymbol{X}$ and $\boldsymbol{y}$: n_k-length vector subset of responses.
for each machine k **do in parallel**
  Compute Intercept by ALLREDUCEing the local weighted average of responses, $\mathbf{1}_{n_k}^{\mathsf{T}} \boldsymbol{y}_{(k)} / n$
  We will now use the intercept-compensated data: $\boldsymbol{y}_{(k)} := \boldsymbol{y}_{(k)} - (\mathbf{1}_{n_k} \cdot \text{ Intercept})$
  Initialize $\mathbf{Weights}_{(k)}$ to p-length vector of 0s ;
  Initialize $\mathbf{CumDiv}_{(k)}$ to p-length vector of 0s ; /* **CumDiv** stands for cumulative divergence */
  repeat
    Use MPI's ALLGATHER across each machine to share all $\mathbf{Weights}_{(k)}$ with all machines ;
    $\mathbf{CumDiv}_{(k)} := \mathbf{CumDiv}_{(k)} + c \sum_{k' \neq k} \left[\mathbf{Weights}_{(k)} - \mathbf{Weights}_{(k')} \right]$;
    for $i \in \{1, ..., p\}$ **do** /* Each weight index */
      /* $\mathbf{PartErr}_{(k)}^{(-i)}$ is the partial residual error, not including the contribution of the ith predictor. $\mathbf{Weights}_{(k)}[\![i']\!]$ means i'th element of $\mathbf{Weights}_{(k)}$; $\boldsymbol{X}_{(k)}[\![, i']\!]$ means i'th column of $\boldsymbol{X}_{(k)}$. */
      $\mathbf{PartErr}_{(k)}^{(-i)} := \boldsymbol{y}_{(k)} - \sum_{i' \neq i} \left(\mathbf{Weights}_{(k)}[\![i']\!] \times \boldsymbol{X}_{(k)}[\![, i']\!] \right)$;
      let $\mathcal{S}$ be the soft thresholding operator, where $\mathcal{S}(z, \mu) = \text{sign}(z) \times \max\{|z| - \mu, 0\}$;
      $\mathbf{Weights}_{(k)}[\![i]\!] := (2c(m-1) + ||\boldsymbol{X}_{(k)}[\![, i]\!]||^2)^{-1} \times$
      $\mathcal{S}\left(\boldsymbol{X}_{(k)}[\![, i]\!]^{\mathsf{T}} \mathbf{PartErr}_{(k)}^{(-i)} + \left(c \sum_{k' \neq k} \left[\mathbf{Weights}_{(k)} + \mathbf{Weights}_{(k')} \right] - \mathbf{CumDiv}_{(k)} \right) [\![i]\!], \frac{\lambda}{m} \right)$
  until *convergence of* ***Weights***;
  return *Intercept,* $\boldsymbol{Weights}_{(root)}$;

Here, $||...||_2$ denotes the ℓ_2-norm, and $||...||_1$ denotes the ℓ_1-norm.

3.3.4 Parallelizing neural networks

In their 2008 work, Dahl, McAvinney, and Newhall outline a data-parallel approach for distributing neural network computation known as Pattern Parallel Training (PPT) [16]. Motivated by this work, we extend and modify the description in order to suit the unique facets of the CEP's dataset.

The distributed neural network derivation is modeled in Figure 4. This network model assumes m total available cores, and the variable $1 \leq k \leq m$ is used to refer to an arbitrary

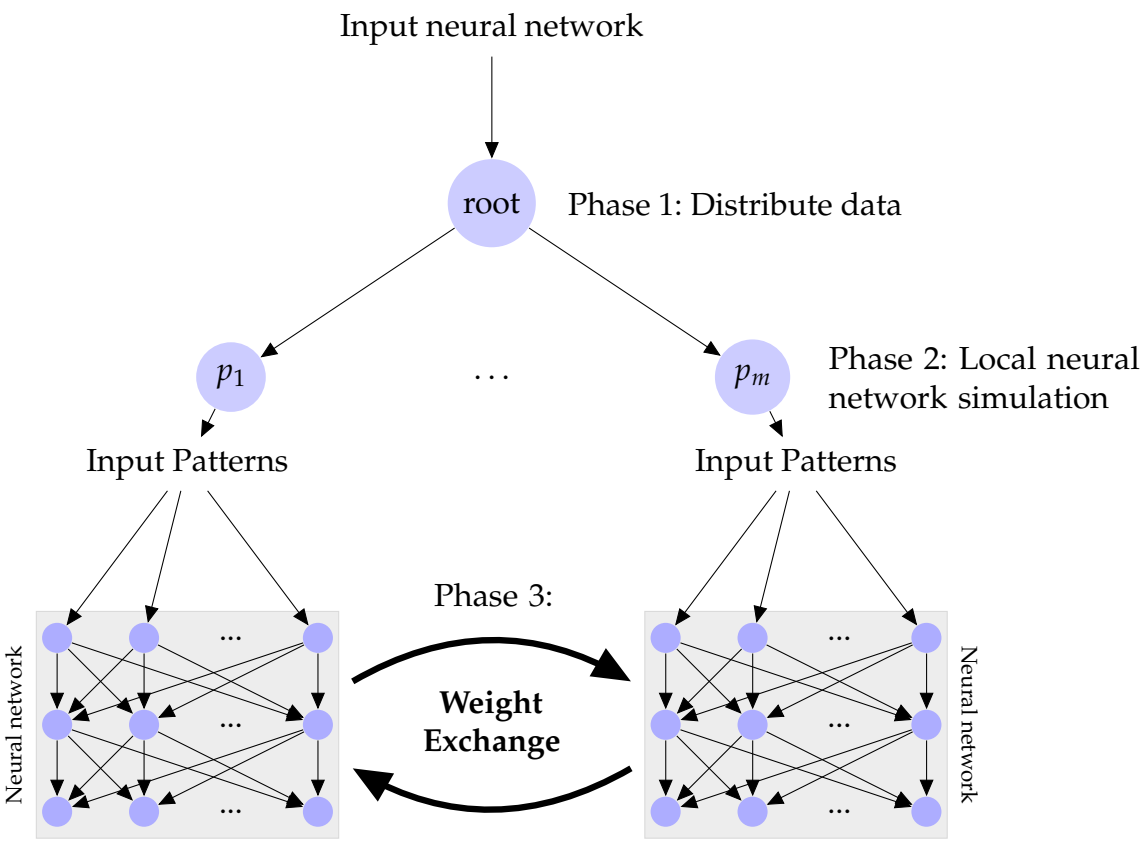


Figure 4: Distributed neural network architecture

core.

Phase 1. This algorithm works by modeling identical neural networks on each of the m cores. We accomplish this by reading in an input neural network on the root core. If running this algorithm for the first time, this neural network can be initialized offline using any desired parameters. Otherwise, this neural network can be a partially-trained neural network, allowing a single network to be trained over multiple executions of this algorithm. In this algorithm, the input neural network and the entire dataset $\boldsymbol{X}$ are replicated on all cores. Note that $\boldsymbol{X}$ must be normalized to be between 0 and 1 in order for the neural network to work correctly. These structures can be replicated easily using MPI's BROADCAST, after the data has been read from disk. This ensures that the entire neural network structure and initialization weights are replicated exactly on each core.

Phase 2. Each core selects a random subset $\boldsymbol{X}_{(k)}$ of the data (these do not have to be disjoint subsets). The size of the data per core, $|\boldsymbol{X}_{(k)}|$, is a parameter to this algorithm. For each core index k, the neural network on core k computes—but does not apply—the batch weight updates for inputs $\boldsymbol{X}_{(k)}$.

Phase 3. Using MPI's ALLREDUCE, sums of all batch weight updates are sent to all cores in the network. Each core then updates its recorded original weights from before the last training iteration by adding these batch weight updates (scaled by the learning rate), with momentum applied from the previous run of Phase 3. We then repeat from Phase 2 with a new random subset.

Final compute. The above sequence of phases is repeated as long as desired. Note that, after the weights exchange, the state of the neural network on each core is identical. Therefore, we can simply use the neural network on one of the cores at the end of the process as the output of the algorithm. This can also be used to output incremental trainings of the neural network during the algorithm. We can estimate the global mean squared error rate by averaging together the local squared error rates across all cores when performing the ALLREDUCE.

Using the algorithm described above is limiting for two reasons. We discuss these and propose solutions below.

1. **Batch weight updates.** This algorithm uses batch weight updates, although it has been shown that incremental neural network training will asymptotically outperform batch training with enough data [36]. In our dataset, we found that using batched weight updates significantly slowed down neural network convergence time. We implemented the alternative described below.

   Instead of computing batch updates, before each training cycle we record the current

weights. Then, we perform *incremental* learning for all of the data on the local core. When ALLREDUCEing the weight updates to all cores, what we communicate is the difference between the final weights and the weights before training. We update the network's weights to be the original weights plus these ALLREDUCEd weight updates.

This has the major conceptual advantage that it allows the network to descend the contour gradient during the training process, rather than only when cross-core weight updates are applied. This results in faster convergence and significantly reduces the chance that the neural network will fall into a shallow local minima.

2. **Weight updates overshoot.** From testing, we observed that certain pathological subsets of the data would result in weight updates on one of the cores that would dominate the weight values. Applying these weight updates would result in a network weight divergence that would cause the network to 'fail' by applying larger and larger weight changes to try to find a reasonable set of weights. Although this event is rare for any individual subset of the data, since we are applying so many distributed weight updates, its occurrence was likely in our dataset. We found that a simple but effective remedy to this problem was to divide the weight updates by the total amount of data being trained over on that iteration, which tended to prevent the weight updates from ever dominating the current weights.

3.3.5 Large data considerations

The above algorithms are effective, distributed architectures for performing machine learning over modestly sized molecular datasets. However, these approaches run into data issues when being run over the large dataset of 1.8 million molecules used in this paper. In the matrix representations of the molecules, some of the representation types had a memory footprint of more than 20GB on disk.

Loading the entire dataset into memory is unfeasible for these representations. Although Odyssey can support up to 256GB per 64 cores, this means that we cannot replicate the entire dataset on each core. For linear regression, this is a simple task of having each core read only the segment of the dataset that it is responsible for (preprocessing can be used to divide the entire dataset into the appropriate segments).

However, for parallelizing neural networks, we run into the difficulty that, according to the algorithm outlined above, each core is responsible for random subsets of the entire dataset for each epoch. We propose the two solutions below for this problem.

1. The first solution is basic but effective. Each molecule's representation can be written to disk as a separate file. Then, at the beginning of Phase 2, each core selects a random subset of the files and loads them as its working set. The runtime of this approach is dependent upon the file I/O time. This approach has the benefit that it can scale to arbitrarily large datasets, independent of the number of available cores: since the entire dataset doesn't need to be read for each epoch, the cores can be made to read a small enough random fraction of the data so that molecular data can be fully stored in memory.
2. The second solution is to partition the entire dataset equally across the cores. A possible downside to this is that, during training, each core will descend along the gradient for the same inputs. This could cause the algorithm to be biased towards a specific local minima if, for instance, the inputs on one of the cores result in weight updates that consistently dominate the other weight updates. However, in practice and for the large dataset used in this paper, this has not shown to be an issue. A second limitation of this approach is that it does not scale independently of the number of cores: each core must be able to store its fraction of the dataset in RAM.

There are certainly more advanced techniques available to counteract this problem, such

as using a live database to store the molecule data, but these kinds of services were not available for this project.

3.4 Data pipeline

This section discusses the full data pipeline used when processing the entire dataset in this project. The full data pipeline is outlined in Figure 5.

Parse the input record. The raw input record contains entries for 1.8 million molecules. Although the actual input record is somewhat more sophisticated than portrayed in Figure 5, from the records we can compute the HOMO and LUMO values and their gaps. The record itself is quite small, since the 1.8 million molecules are all represented in SMILES notation. As a result, we can use a serial parser to parse the input record in a series of tuples of $\langle$SMILES, HOMO-LUMO gap$\rangle$. We partition this list of parsed tuples in 512 lists of tuples and write them to the file system.

Compute predictors in parallel. The next phase computes predictors for each of the molecules in an embarrassingly parallel way using the methods outlined in Section 3. Since we have 512 records on disk, we can compute our desired predictors over the dataset using 512 different cores. Specifically, for each molecule, we compute the following predictors:

1. The 59 real-valued ChemAxon features listed in Table 2
2. Full sorted Coulomb matrix
3. Coulomb matrix eigenvalues in decreasing order
4. Full binary adjacency matrix
5. Binary adjacency matrix eigenvalues in decreasing order
6. Full n-ary adjacency matrix
7. n-ary adjacency matrix eigenvalues in decreasing order

Reduce to compute data statistics. Now that the predictors have been computed for all

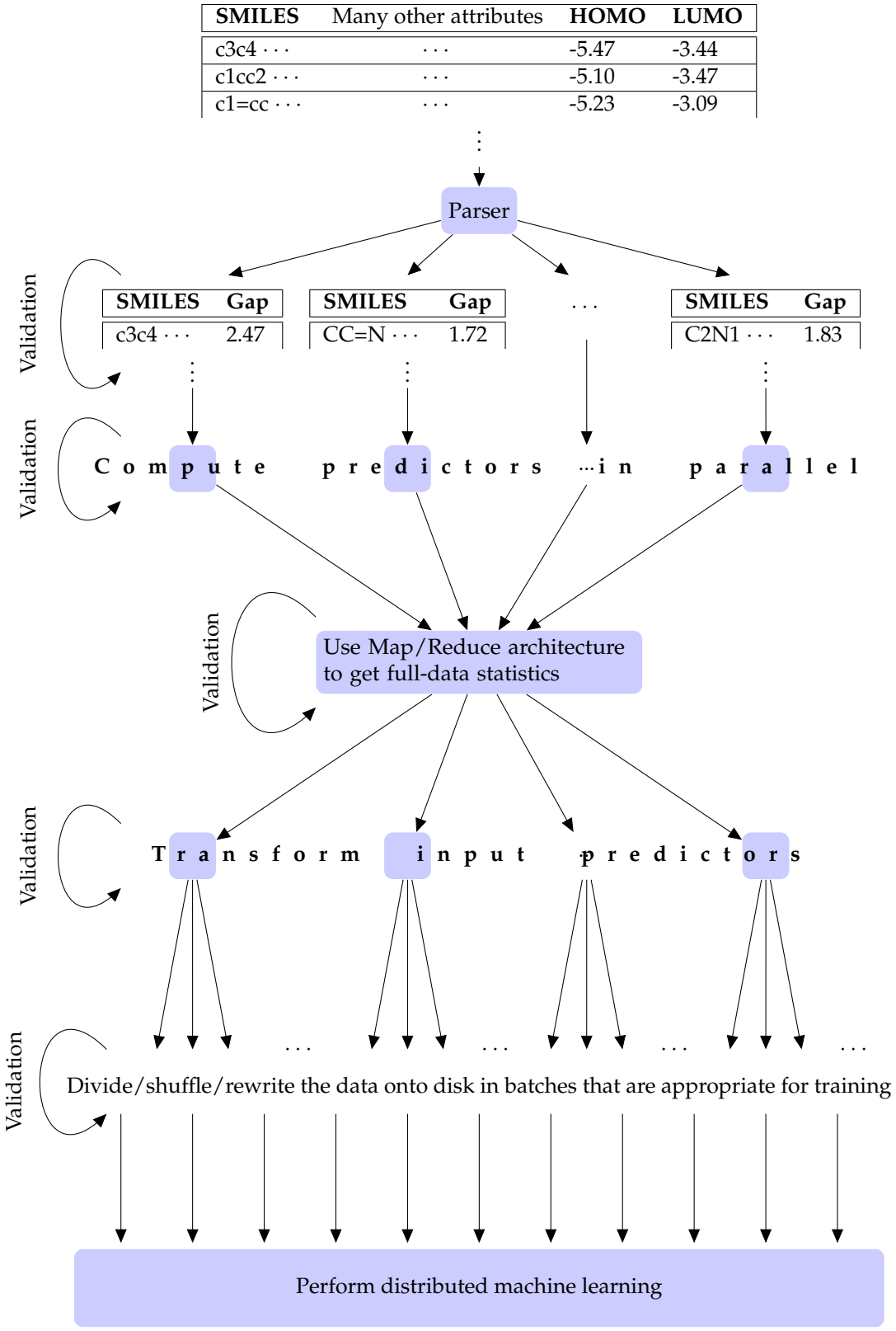


Figure 5: The data pipeline for the full dataset

molecules, we need to compute a few statistics from the data.

1. **Global mean and variance of each predictor.** This is needed to center and rescale the predictors for linear regression.
2. **Maximum dimensions for each representation type.** We need to determine the maximum dimensions for each representation type. For eigenspectra, this means counting the max number of eigenvalues, and for matrices, this means counting the max number of entries in the matrix (as the matrix will be flattened during learning).
3. **Get high/low values for each predictor type.** For instance, for the matrix representations, we need to know the high and low values for each position of the matrix across the entire dataset. We also need to know the high and low values for the response variable. These will be used to normalize the inputs for neural network regression.

These values can be efficiently computed using a distributed Map/Reduce framework.

Transform predictors. As suggested in the previous section, the representation types as computed may have different numbers of dimensions. This is a result of different molecules having different numbers of atoms. We make an embarrassingly parallel pass through the data in order to clean this up. During this pass, we pad eigenvalue lists or matrix dimensions with a number of zeros so that each input for a given representation type has the same number of dimensions. For the full dataset, this means padding all matrices to be shape 35×35 and padding all eigenspectra to be lists of length 35.

For the linear regression approaches, we want each predictor type to be centered and rescaled to have a mean of 0 and variance of 1. This ensures that the regularization levels are interpreted correctly and penalize each of the weights independently of the scale of the predictors (Figure 7 shows that the scales of the predictors vary significantly). For the neural network approaches, we want each predictor and the response variable to be

normalized to between 0 and 1. These transformations can be done during this phase.

Divide and rewrite data. This phase is used to rewrite each input in preparation for learning. This can involve rewriting each datum as a separate entry on the filesystem, or dividing them into batches suitable for the different learning approaches.

Perform machine learning. At this point, our dataset is in the appropriate form to perform regression, and we can use either linear regression or neural networks, as discussed in Section 3.3, to perform molecular feature learning.

Validation at each step. Note that, at each step that involves distributed processing, a post-processing validation step must be used to ensure the integrity of the computation. As was found during experimentation, cores would sometimes report success despite having actually been unsuccessful in computing their desired value due to, e.g. network communication failure or I/O failure. This would potentially cause a core to receive an empty list or fail to actually write a file to the filesystem. Although these failures were infrequent, validation was necessary to rectify the instances in which they did occur. Validation took a different shape for each different phase. In general, when writing to the filesystem, the validation phase would ensure that the number of files that we expected to write were written correctly. When transforming the inputs, we would verify that each input had a specific length. Of course, since the validation phases had to be run in parallel, it's possible for these phases to fail erratically as well, but we had to accept this shortcoming as part of the distributed nature of this project.

4 Results and analysis

4.1 Data exploration

This section examines some of the basic features of the input data before any machine learning was done.

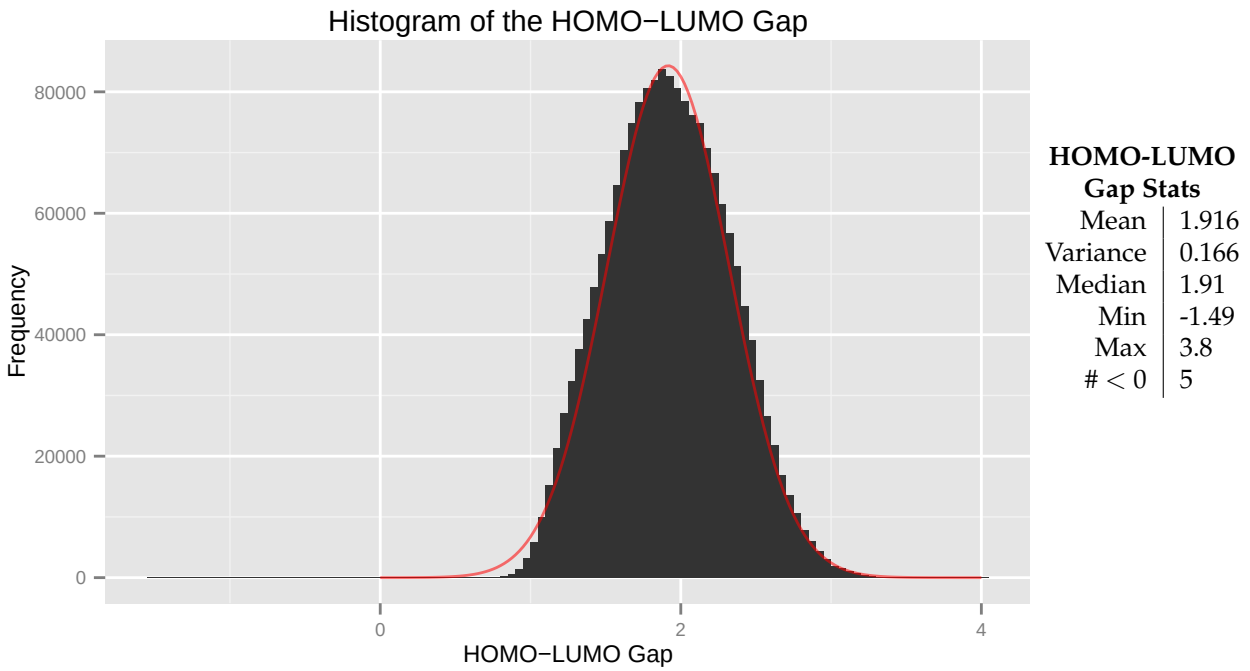


Figure 6: Empirical HOMO-LUMO distribution with statistics. The inferred normal distribution is overlaid.

4.1.1 Empirical distribution of the response variable

Figure 6 shows the histogram of the HOMO-LUMO gaps observed in the input dataset. From the histogram, the distribution of HOMO-LUMO gap values appears to be approximately normal. In the diagram, we have also overlaid a normal distribution curve to the data, with mean and variance set to the mean and variance observed in the response variable. This inferred distribution appears to be a very good fit, demonstrating that the empirical distribution is only very slightly positively skewed.

This is beneficial for several reasons. This distribution tells us something about the response values that we are predicting. From a Bayesian perspective, we could conceptualize this empirical distribution as a prior on our model. Since this empirical distribution is very well-modeled by the normal distribution, we can instead use the normal distribution as a prior for the response variable, which would lead to appealing predictive properties. Furthermore, the fact that the full-data distribution is well-described by a single distribution

increases our confidence that each response variable is individually modeled by a distribution of the same form over its predictors. This is a nice property, because it simplifies our learning approaches. If the distribution of the response variables were to be many-modal, for instance, we would expect standard learning techniques to underperform, and we would have to consider more sophisticated approaches. Finally, this empirical distribution informs us about the types of values that we should be predicting, and gives a sense about what HOMO-LUMO gap values we would tend to consider high and low in our dataset.

One strange observation from the empirical distribution is that there are five negative HOMO-LUMO gaps values. The gap is computed by subtracting the highest occupied molecular orbital from the lowest unoccupied molecular orbital, and so negative values are not valid. This is likely a result of incorrectly computed data in the original dataset, and so these values can be thrown out.

4.1.2 Features distribution

Figure 7 shows boxplots of the feature values. This can help give a sense of the molecular covariates that we're examining in a more interpretable way.

It is clear from these boxplots that the features can take on many different ranges of values. We've identified four bands of feature values, and these correspond to the four groupings in Figure 7. Each of these bands covers approximately a different order of magnitude of values: 1-10, 10-100, 100-1000, 1000-10000. Interestingly, these bands also somewhat separate different types of features. The 1-10 band is comprised mostly of counts of different molecular features, such as different types of bonds and rings. The 10-100 band contains counts, but also has several features relating to polarity and energy (realize that $a(xx), a(yy)$, and $a(zz)$ are polarities). The 100-1000 band contains a mix of energies and areas (the ASA features are areas). Finally, the 1000-10000 band has three features, two of which are indices that summarize the molecular density, and a volume.

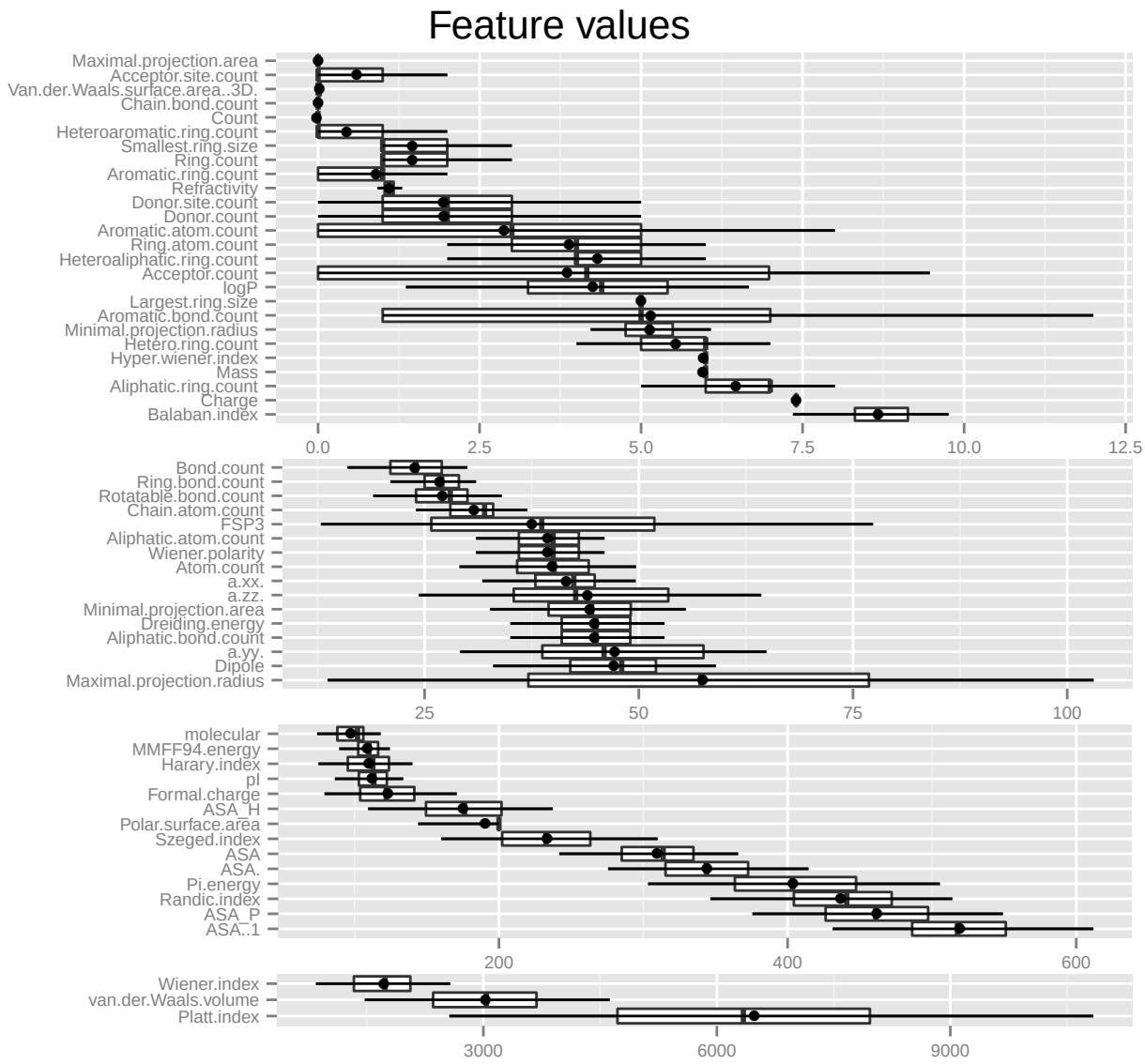


Figure 7: Distributions of feature values. Boxplots Indicate 5%, 25%, 50%, 75%, and 95% quantiles, points indicate means. Features are sorted by their medians.

Realizing the different bands of features is important for future study. These bands may be exploited in order to perform dimensionality reduction or to develop additional learning approaches tailored to each feature, group of features, or band. Examination of the bands after machine regression may also help to identify which groups of features are the most useful predictors for the response variable.

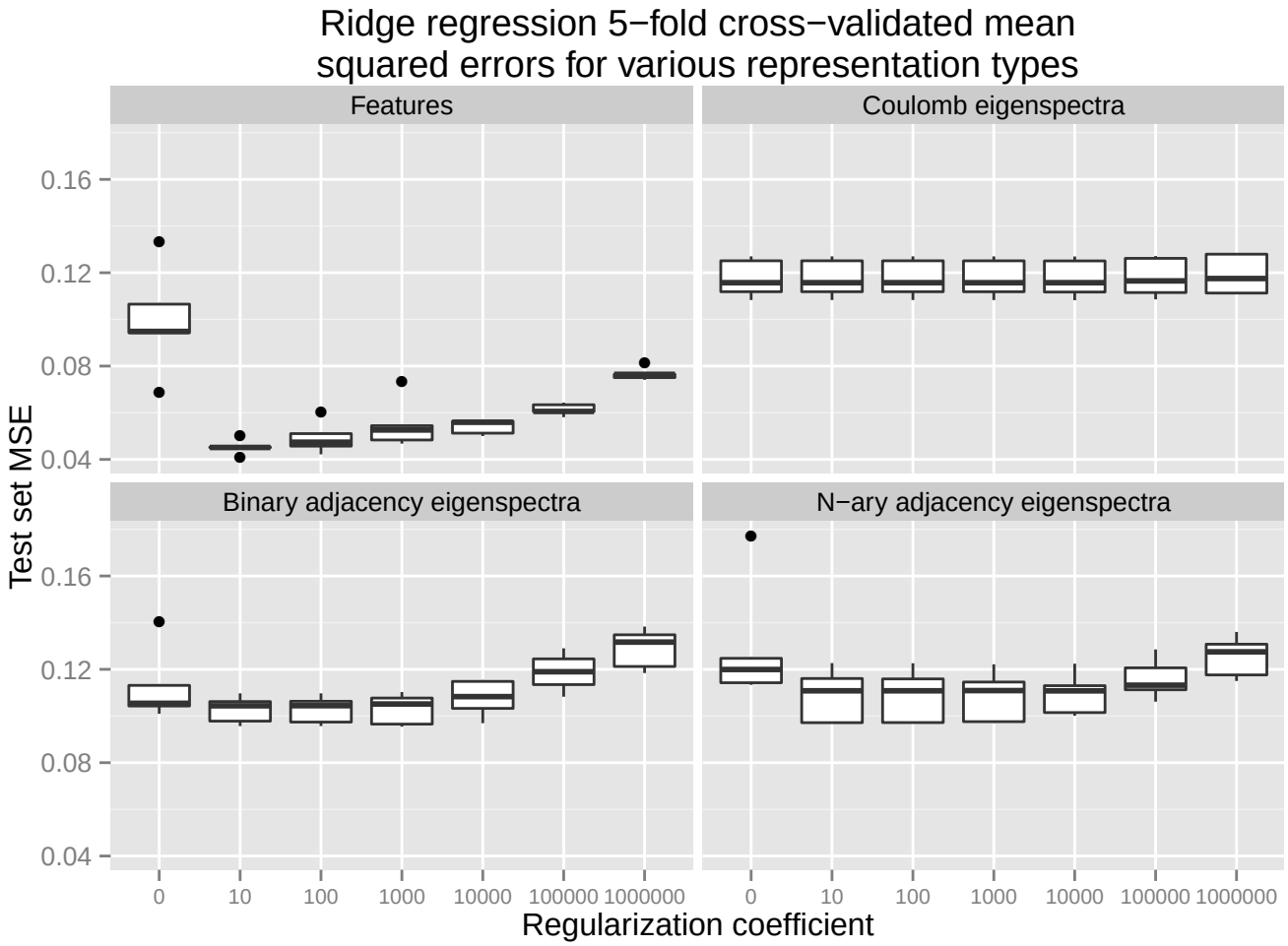


Figure 8: MSEs for OLS and ridge regression at different regularization levels. $\lambda = 0$ corresponds to OLS in the above figures. The boxplots show the test set MSE for each of the five cross-validation folds used.

4.2 OLS and ridge regression

We performed ℓ_2-regularized ridge regression by choosing a regularization level λ on the weights $\hat{\boldsymbol{\beta}}_1$ and solving for them as

$$\hat{\boldsymbol{\beta}}_1 = \left(\lambda \mathbb{I}_p + \overline{\boldsymbol{X}}^{\mathsf{T}} \overline{\boldsymbol{X}}\right)^{-1} \overline{\boldsymbol{X}}^{\mathsf{T}} \overline{\boldsymbol{y}}.$$

We used different orders of magnitude of λ between 10 and 1 million to approximate the optimal regularization penalty. Note that when $\lambda = 0$, this is the same as OLS regression. A plot of the resultant cross-validated MSEs is listed in Figure 8. For each regularization

level, we split the data into five folds and ran five tests, knocking out one fold in each test to use as the test set. After performing learning over the training set, we performed prediction over the test set, and recorded the mean squared errors in the figure.

From the results, we note several observances. It seems apparent that, of all the representation types, the features representation performed the best. While most other representations appear to have a best test set median MSE on the order of 0.1, this representation had a best test set median MSE of under 0.05 for a regularization level of $\lambda = 10$. The two adjacency representations appear to perform analogously, with the n-ary representation in general performing slightly worse. The Coulomb eigenspectra representation seems to perform slightly worse than the adjacency representations. In terms of regularization, we note that introducing even a small amount of regularization is very important. For all representation types except Coulomb eigenspectra, the MSE for $\lambda = 0$ was significantly higher than the MSE for $\lambda > 0$. However, realize that the optimal λs, which for all models appear to be between 10 and 100, are very small considering the number of inputs. This suggests that there may be only a modest amount of overfitting, which is what is expected given the large size of the dataset. Interestingly, we notice that there is virtually no influence of λ on the MSE for the Coulomb eigenspectra representation. Most likely, this means that the observed data very strongly implies a certain set of weights despite even large regularizations. Further, there is very little difference between the OLS and ridge regression MSEs for the Coulomb eigenspectra, which means that there is essentially no overfitting. This hypothesis is corroborated by observations with LASSO regression in Section 4.3.

It is not surprising that the features representation had the best performance. The features directly model attributes of a molecule, while the eigenspectra are summarizations of larger matrices. The weights for the individual standardized features are displayed in Figure 9.

Figure 10 shows the accuracies of the ridge regression models on the training set. For the features and Coulomb eigenspectra representations, these were generated using $\lambda = 10$, and for the adjacency representations, these were generated using $\lambda = 100$. From the plots, we can see that the predictions are decent, but not excellent. The features representation plot looks very promising, with most of the data lying closely along the actual = predicted line. However, for the other representations, the data has somewhat higher variance. In particular, the Coulomb eigenspectra representation appears to be particularly poor, with strange clusters of significantly under-approximated points. The adjacency representations have more tolerable accuracies, but seem to have strange bands of misidentified points for certain predicted values.

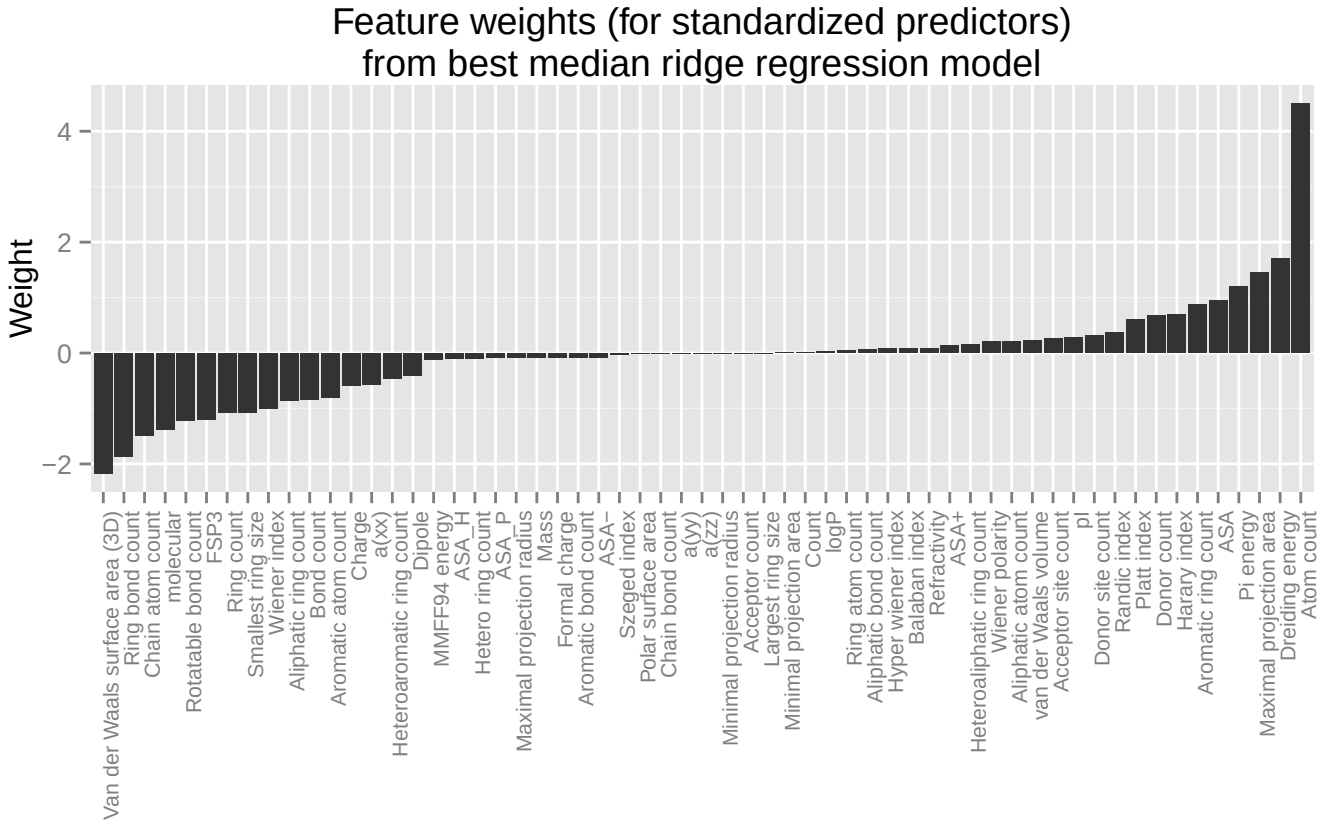


Figure 9: All feature weights for the cross-validation fold with the median best performance at the $\lambda = 10$ regularization level for ridge regression.

We can also get a sense of model accuracy by examining prediction residuals. Figure 11 shows histograms of the differences between the actual HOMO-LUMO gaps and their predicted values. As expected, the features representation results in the narrowest band of residual values. However, importantly, we realize that the residuals for *all* of the models appear to resemble normal distributions centered around zero, with worse models having higher variance in their residuals. The fact that the residuals look normal is important, because it means that there is little improvement to be made through transforming the input data. If these distributions were not normal, then we may be able to apply transformations

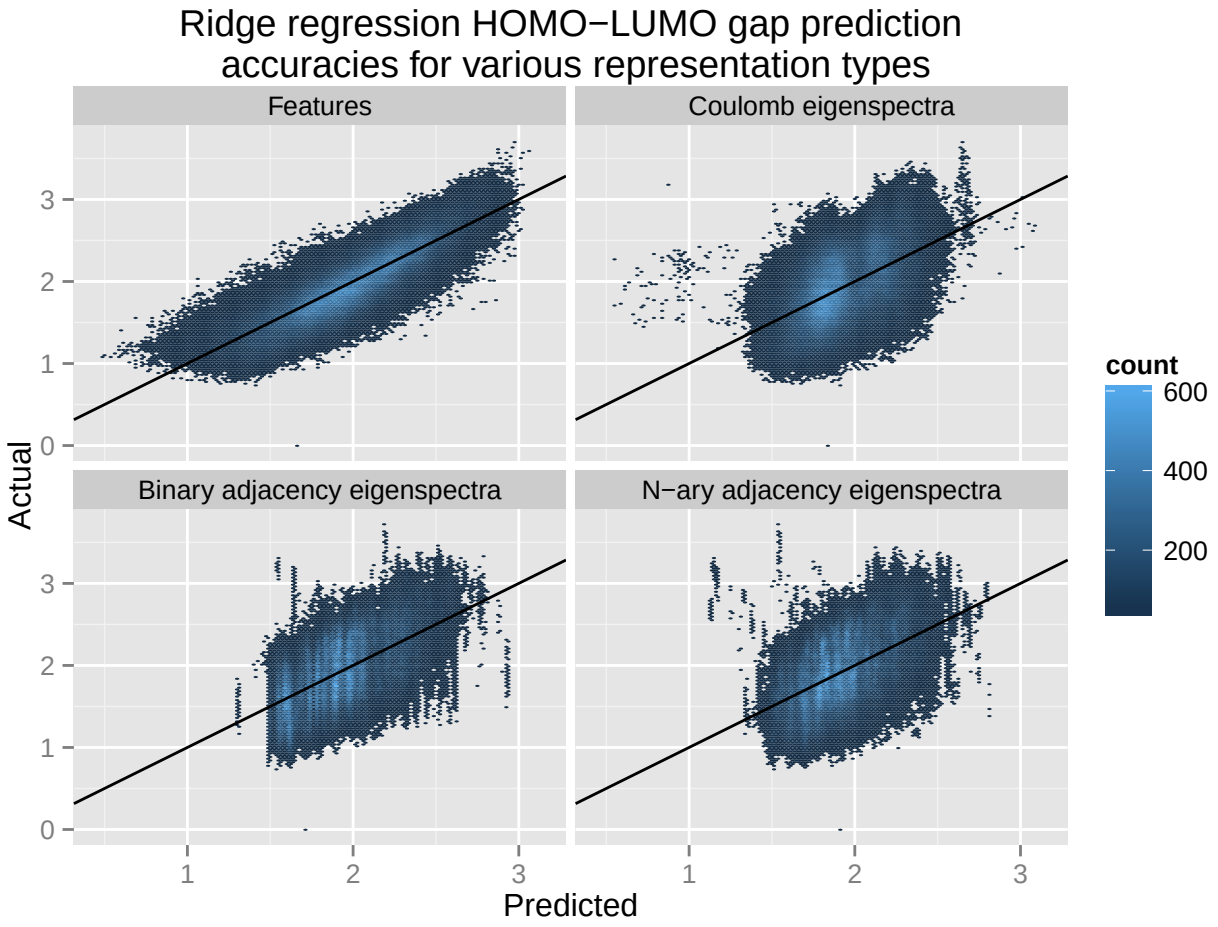


Figure 10: Scatter plot of actual value versus predicted value using the cross-validation fold with the median best MSE under ridge regression. For the features and Coulomb eigenspectra representations, these were done with $\lambda = 10$ for the regularization level, and for the adjacency eigenspectra, these were done with $\lambda = 100$. The ideal actual = predicted line is overlaid.

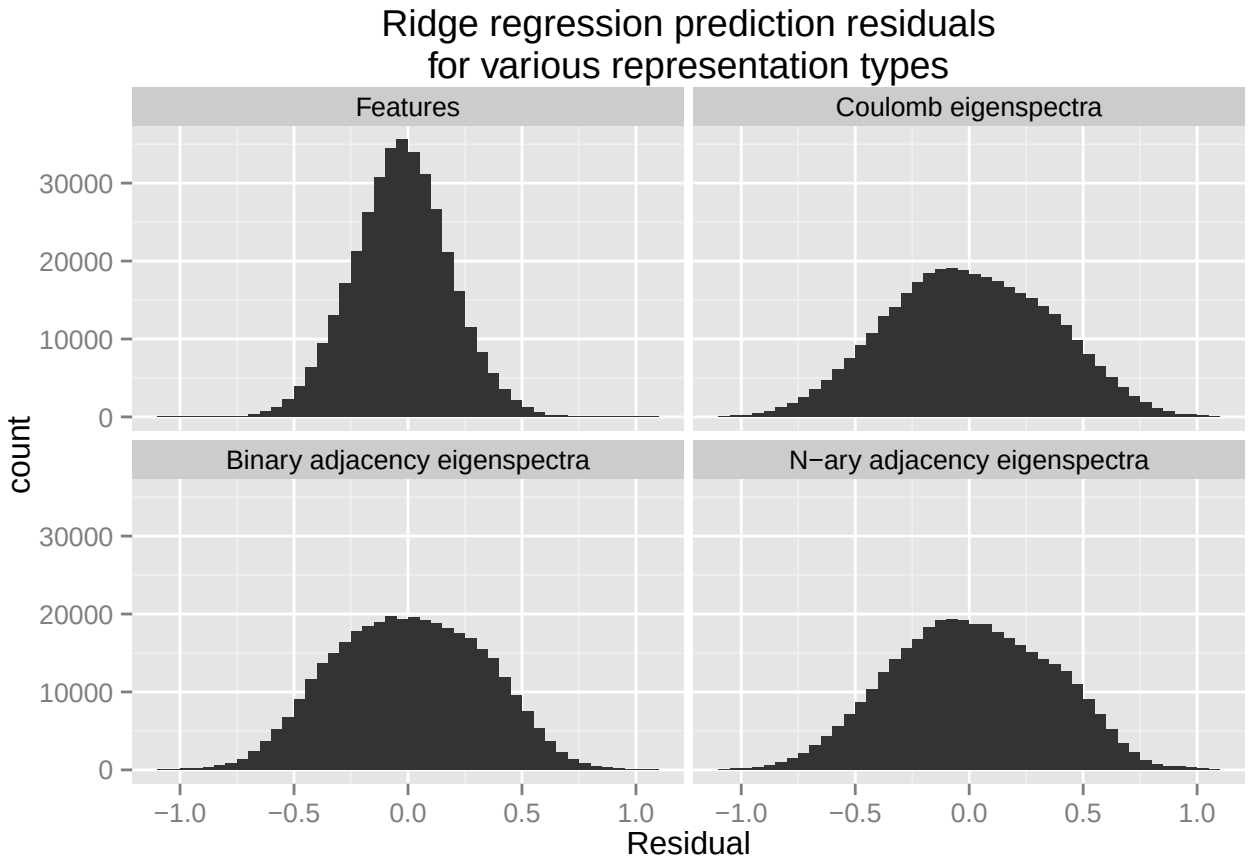


Figure 11: Histogram of prediction residuals using the cross-validation fold with the median best MSE under ridge regression. For the features and Coulomb eigenspectra representations, these were done with $\lambda = 10$, and for the adjacency eigenspectra, these were done with $\lambda = 100$.

that the regression could more easily fit; however, these residuals show that we are probably predicting about as well as we can using linear models.

4.3 LASSO regression

We performed ℓ_1-regularized LASSO regression by iteratively solving the minimization problem

$$\hat{\beta}_0, \hat{\boldsymbol{\beta}} = \underset{\beta_0, \boldsymbol{\beta}}{\arg\min} \frac{1}{2} ||\boldsymbol{y} - \mathbf{1}_n \beta_0 - \boldsymbol{X}\boldsymbol{\beta}||_2^2 + \lambda ||\boldsymbol{\beta}||_1.$$

We used different orders of magnitude of λ between 100 and 10 million to approximate the optimal regularization penalty. A plot of the MSEs is listed in Figure 12. The cross-validation procedure is the same as that used in the ridge regression experiments (Sec-

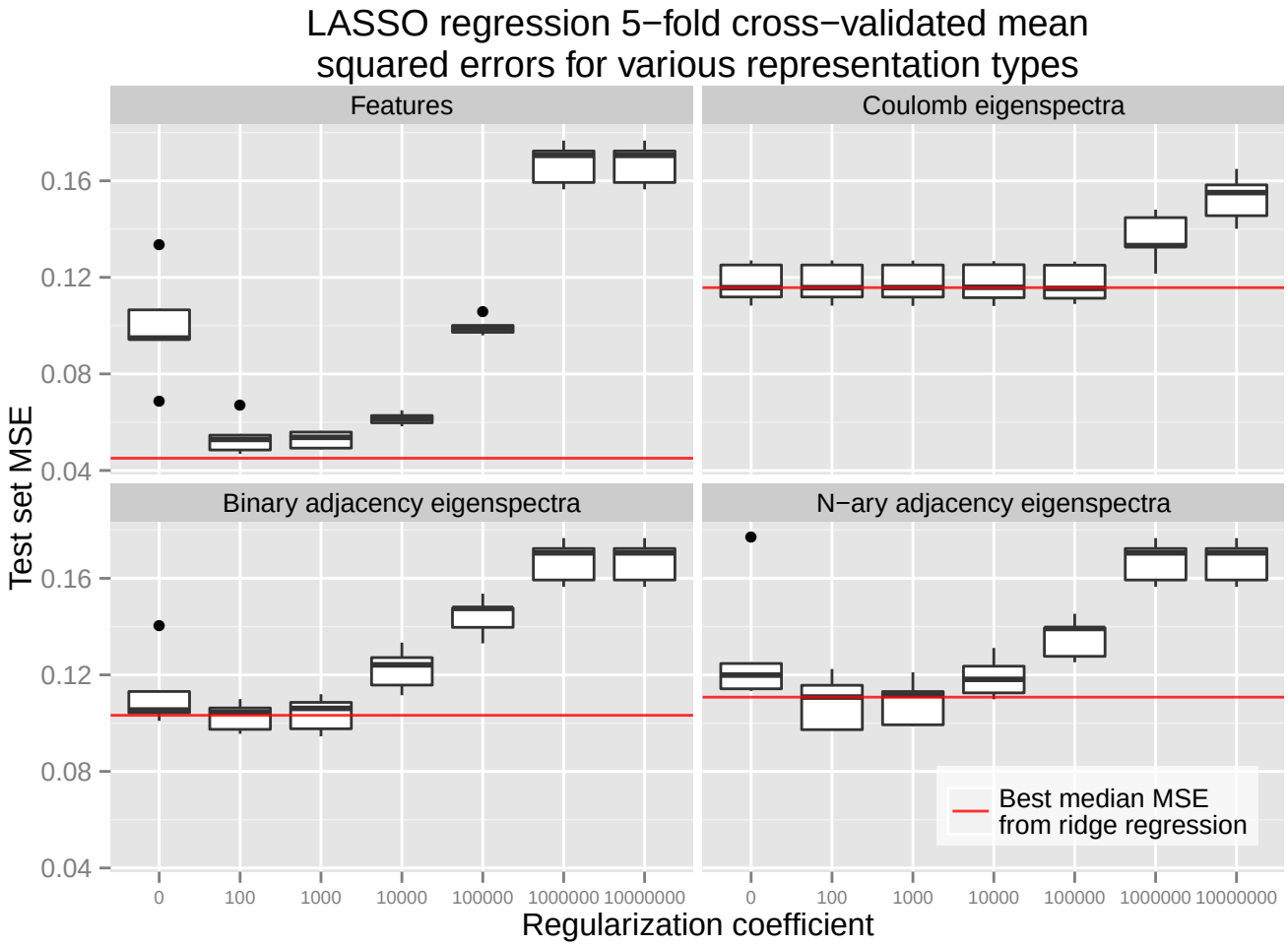


Figure 12: MSEs for LASSO regression at different regularization levels. The boxplots show the test set MSE for each of the five cross-validation folds used. The OLS MSEs are also shown for reference.

tion 4.2). The accuracies and residuals looks almost identical to those found with ridge regression in Figures 10 and 11, and so will be omitted.

From the plots, we notice that the optimal LASSO regularization values are all around 100. The optimal observed MSEs for these regularization levels are very similar to their counterparts found with ridge regression. Note that, in both approaches, the optimal regularization levels were very small compared to the size of the dataset. This means that there is likely very little overfitting in the model, and explains why the optimal MSEs achieved by using the two methods are very similar — at low regularization levels, these regressions are essentially performing the same process. The results of the Coulomb

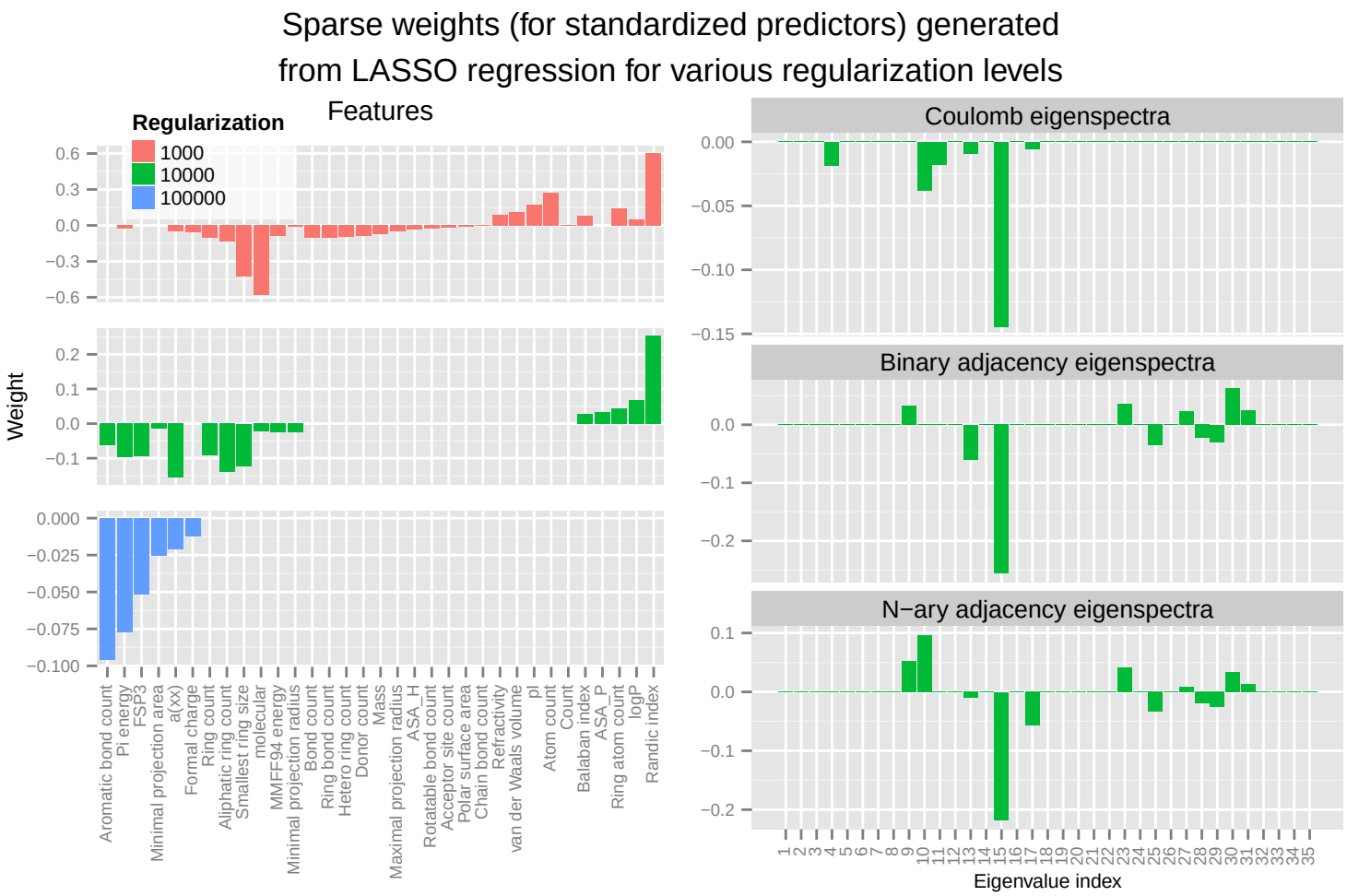


Figure 13: Nonzero feature weights and eigenspectra predictor weights for the cross-validation fold with the median best performance for LASSO regression. We here show the weights for several informative regularization levels for the different representations.

eigenspectra in this experiment corroborate the beliefs that we asserted about its overfitting in Section 4.2: the MSE does not seem to change even for quite large regularization penalties, which means that there is likely very little overfitting and that the observed data strongly implies the observed feature weights.

The sparsity-inducing nature of LASSO regression is useful for performing feature selection. For low regularization levels (such as the ones that led to the best MSE), not much sparsity will be induced. However, for higher regularization levels, LASSO regression suggests a small set of features which may best characterize the data. Some of the more informative

regularization levels are shown in Figure 13. For brevity of presentation, this figure shows the feature weights for three different regularization levels, and the eigenspectra weights for one regularization level each.

From the graphs for features, we see that the nature of the desired features changes as we regularize less and less. At the high regularization level, we see weights on features such as energy, FSP3 (a measure of the fraction of carbon atoms), projection area, and $a(xx)$ (a measure of polarity). These features descriptors that summarize characteristics of the molecule as a whole. At the middle regularization level, some of these features become less important, and we see that ring counts and descriptions of ring size become important. We also see that the Randic index becomes prominent, which is a measure of molecular connectivity. These are descriptors that, when considered together, provide information about the structure of the molecule. At the low regularization level, there are many weights, and these weights are often over low-level characteristics such as counts of various features of a molecule. These observations are as expected: when we restrict the set of weights more, more importance is placed on high-level characteristics that can describe the molecule as a whole. When weights are less restricted, they can focus on lower-level features that differentiate molecules more specifically.

The eigenspectra graphs are interesting in that they all show a large spike for the 15th eigenvalue. We would expect the binary and n-ary adjacency eigenspectra to lead to similar sparse weights. However, the fact that both the Coulomb and adjacency representations share the same strongest weight is peculiar, and worth investigation to determine if this is merely a coincidence. Overall, it is interesting that the models place weights toward the more mid-ranged eigenvalue indices as opposed to the earlier indices. This emphasis suggest that subtle differences between the underlying matrices may be the most important differentiating factors in identifying the HOMO-LUMO gap from a matrix-based representations.

4.4 Neural networks

The neural networks were trained using two different architectures. Both architectures used are listed in Table 3. The types of configurations were chosen so that the number of nodes in each layer was mathematically related to the number of input dimensions. For configuration 1 this was a proportional relationship, and for configuration 2 this was an nth root relationship. We chose the specific number of hidden layer nodes such that there would be neither very many nor very few numbers of nodes in each dimension. From experimentation, we found that too few nodes inhibited learning. Too many nodes compared to the input dimensions resulted in immediate overfitting; within a few epochs, the neural networks would converge prematurely to local minima, and subsequently not benefit from additional training. It has been observed in the literature that using more than two hidden layers often does not improve training performance [37]. In fact, using too many hidden layers will cause the network to explore too many local minima, and could impair overall training if random restarts are not used [38]. The choice of architectures used was limited by the amount of available computation, training, and research time, and interesting follow-up research could certainly explore the differences between a variety of network architectures. Also due to time restrictions, instead of performing cross-validation, we held back 1/10 of the data as a test set, and used the rest of the data for training.

Figure 14 shows the test set MSEs as a function of number of training epochs for the first configuration of neural networks. Figure 15 compares the final test set MSEs after training.

Configuration	**Representations**	**# Layers**	**Number of hidden nodes** (p is # dimensions)
Configuration 1	List types	2	Layer 1: $p/2$, layer 2: $p/4$
	Matrix types	2	Layer 1: $p/8$, layer 2: $p/16$
Configuration 2	List types	2	Layer 1: $2\sqrt{p}$, layer 2: $2\sqrt[3]{p}$
	Matrix types	2	Layer 1: $\sqrt{p}$, layer 2: $\sqrt[3]{p}$

Table 3: Neural network architectures. Note that 'list types' refers both to the eigenspectra representations and the features representation.

Unfortunately, time constraints limited the number of training iterations. In particular, the matrix-type representations took significantly longer to train due to the additional number of dimensions and especially because of the additional number of nodes per hidden layer. Even accounting for this, however, these plots are very informative for providing insight into the comparative performance of the different representation types. From these figures, it is clear that the list-type representations perform significantly better than the matrix-type representations. In fact, the matrix-type neural networks performed on average even worse than the linear regression models.

The accuracy plots in Figure 16 provide additional insight into the neural network performance. It is clear due to the density of points along the predicted = actual line that the features representation performs very well. It is interesting to observe that the accuracy plots of the matrix representations seem to be rotated slightly counterclockwise away from

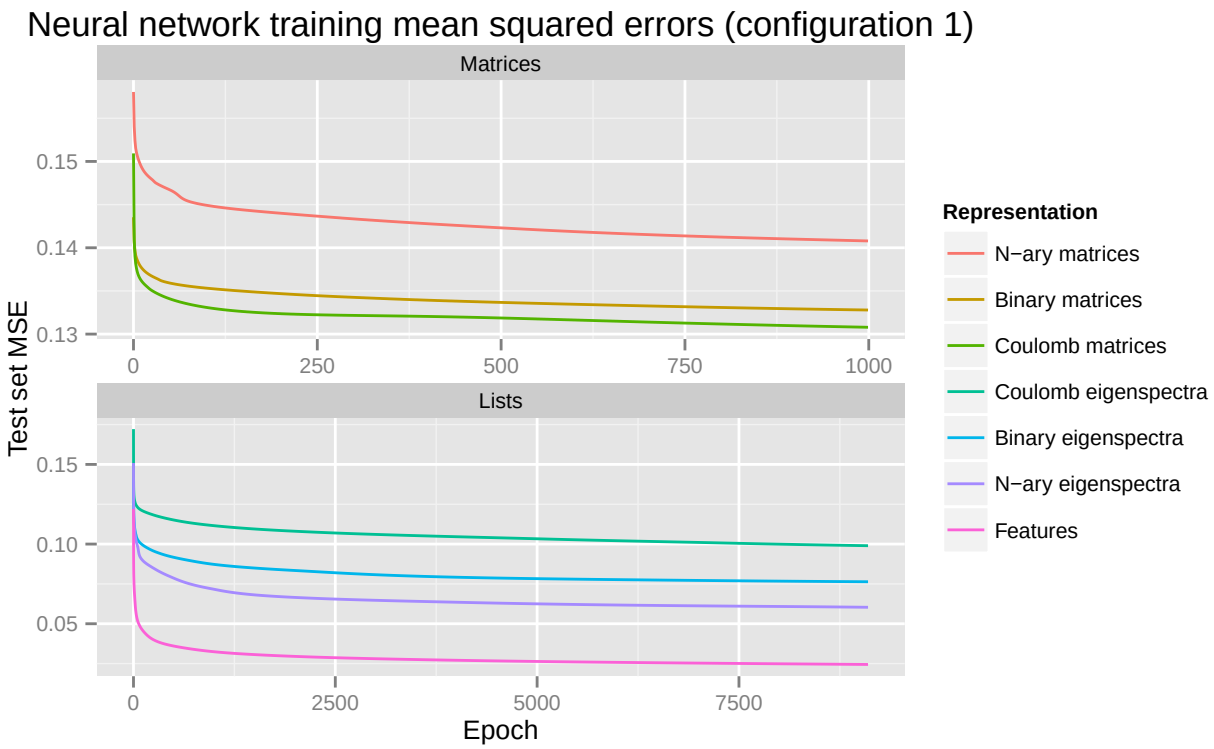


Figure 14: Approximate neural network MSEs during training for neural network configuration 1

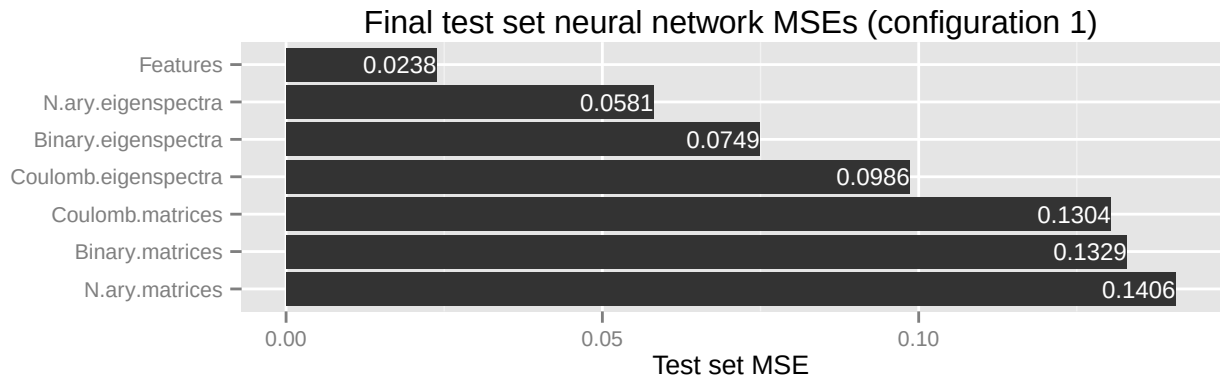


Figure 15: Neural network test set MSEs after training for neural network configuration 1

what may appear to be the optimal values. This means that these neural networks are predicting too conservatively: they often predict molecules with low HOMO-LUMO gaps as having higher HOMO-LUMO gaps than they really do, and they predict molecules with high gaps as having lower gaps than they actually do. This would be particularly bad if we were to try to use these networks to identify molecules with high potential photovoltaic efficacies.

There are two likely causes for the fact that the matrix representations perform worse than the eigenspectra representations. Importantly, we had to sort these Coulomb matrices in order to maintain invariance between permutations of the atoms. However, in doing this, important information that was encoded in the original matrices was lost. Each element of the matrix is no longer interpretable on its own. Neural networks are quite powerful at learning patterns in the input data, but performing this transformation threw out many patterns that could potentially be used for regression. Furthermore, our transformation was arbitrary. As seen in the literature, there are other possible transforms, and in fact any transformation that maintains invariance to permutation of the atoms is valid. What makes one transformation better than another is a difficult question to answer and an area of ongoing research.

Secondly, it is quite possible that there is a substantial amount of overfitting occurring with

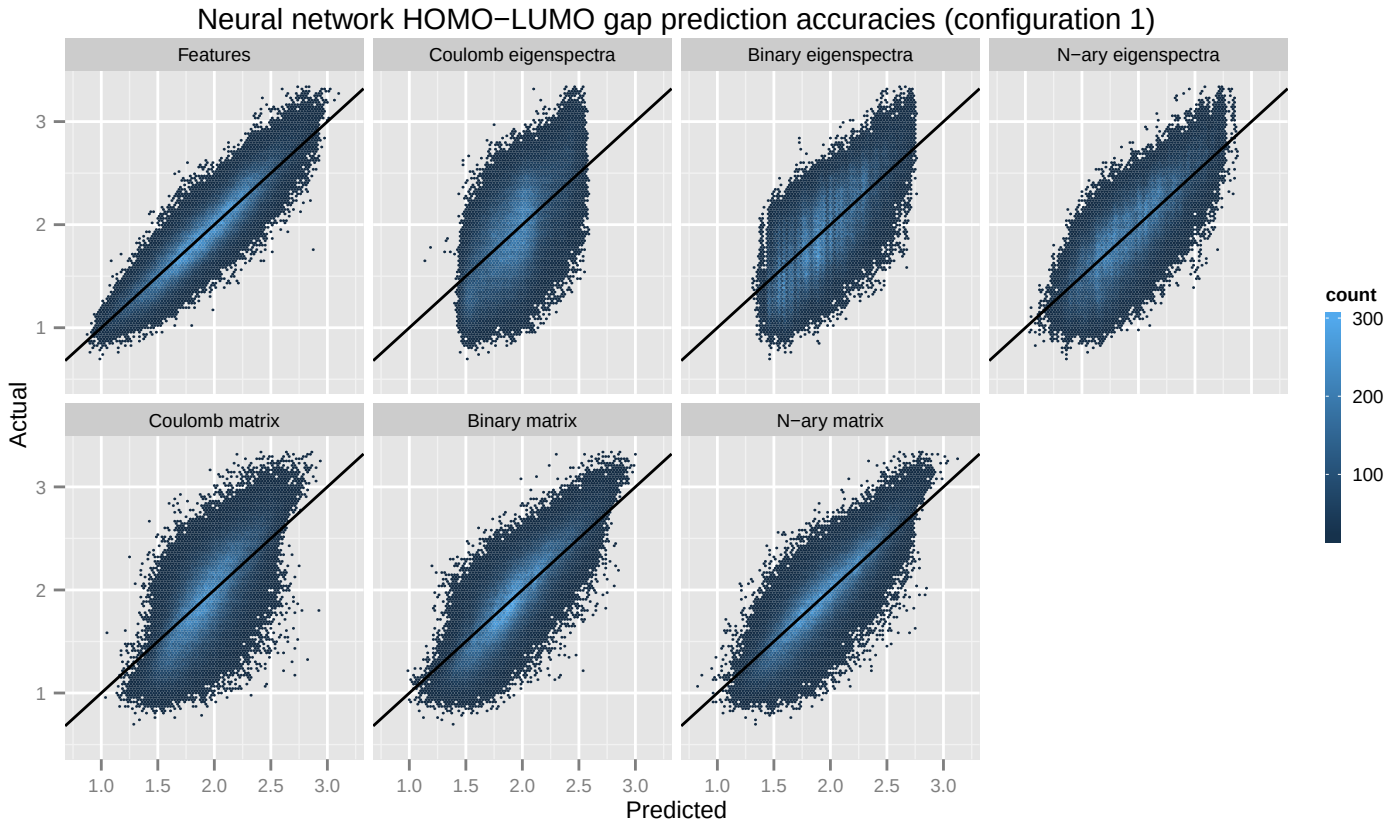


Figure 16: Scatter plot of actual value versus predicted value using on the test set using neural network configuration 1. The ideal actual = predicted line is overlaid.

the matrix-type representations. Overfitting is likely because of the additional number of inputs and the additional number of hidden nodes compared to the list-type representations. Having additional inputs means that there are simply more predictors per input for the neural network to learn from. The network is therefore more likely to learn artifacts due to coincidental patterns in the data. The additional number of hidden nodes means that the network is more likely to wind up in local minima. Since the network has more flexibility in its weight assignments, for a given error residual, each weight can be updated by a smaller amount when compared with the list-type neural networks. In short, the list-type neural networks have more constrained weights, which helps to prevent overfitting.

With these remarks, we can look to the list-type representations to determine relative efficacy. It appears that the eigenspectra representations all perform similarly well, while

the features representation offers the best test set mean squared error. As with linear regression, we suspect that this is a result of the features representation more directly modeling properties of the molecule when compared to the eigenspectra. With neural networks, however, we are able to achieve a significantly lower mean squared error compared to linear regression. This is because there are likely complex interactions between the attributes of a molecule. For instance, it is possible that a molecule would have a high HOMO-LUMO gap if it has either high drieding energy or high MMFF94 energy, but not both. This type of nonlinear relationship can be learned and modeled with neural networks but not with linear regression.

A final remark about this data is that it appears that these neural networks have not yet converged, especially for the matrix-type representations. This means that further significant decreases in MSE may be observed simply by training the neural networks for

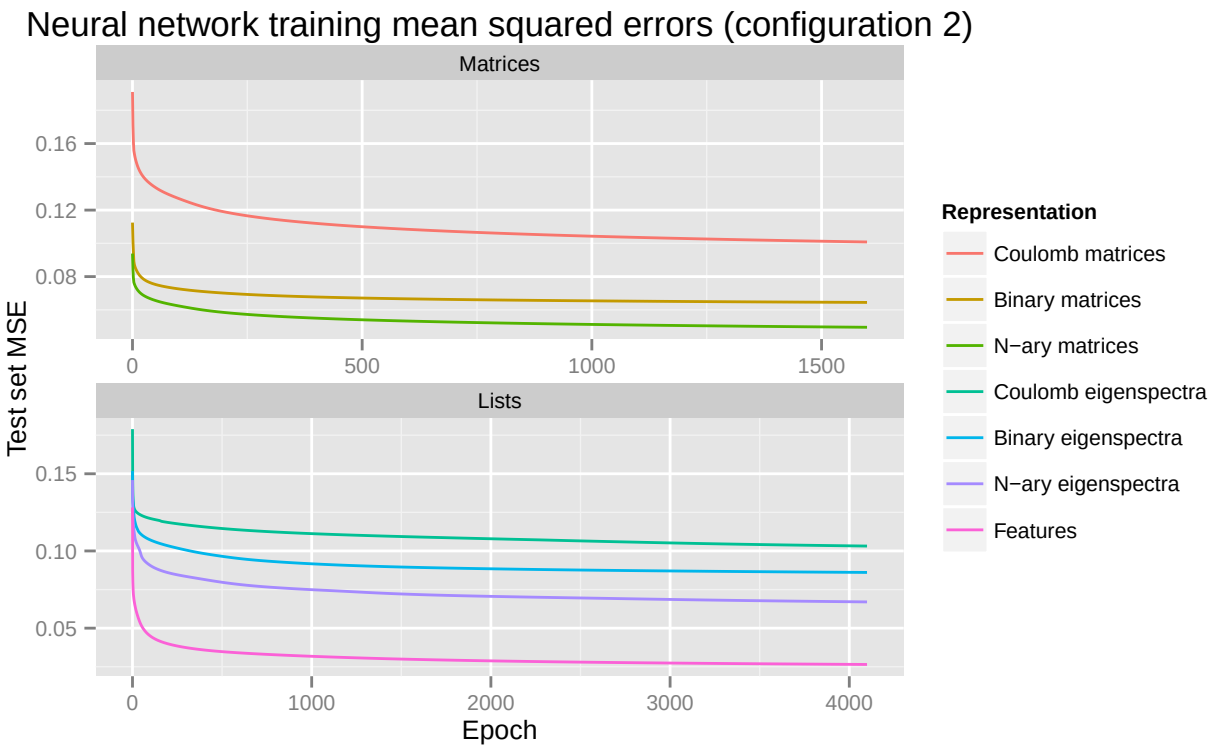


Figure 17: Approximate neural network MSEs during training for neural network configuration 2

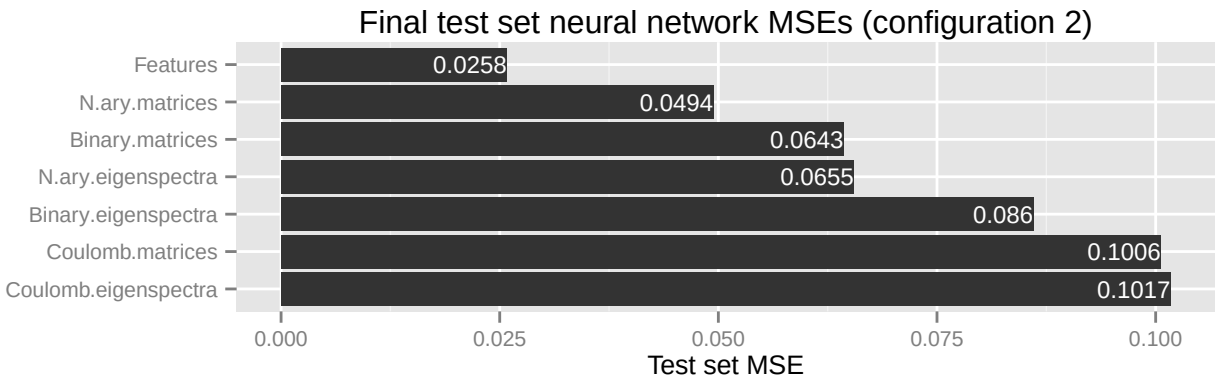


Figure 18: Neural network test set MSEs after training for neural network configuration 2

longer. The current MSE data is complete enough to draw conclusions about the relative performance of the different models. However, additional training offers the potential for significantly improved prediction, and would be desirable in the implementation of a large-scale quantum-molecular classifier.

We repeated these experiments for the second neural network configuration. Figure 17 shows the test set MSE during training, and Figure 18 shows the final MSEs achieved at the end of training. An important difference between this configuration and the previous one is that this configuration provides fewer hidden nodes to the list representations, and significantly fewer nodes to the matrix representations.

Overall, we observe significant improvement for all matrix-type representations. While the features representation performs about the same as in configuration 1, the eigenspectra representations perform slightly worse.

These results are significant because they reveal the importance of the neural network structure. The matrix-based neural networks were able to perform better in this configuration than their eigenspectra counterparts did in the first configuration. This is strong evidence that overfitting is a major concern in this domain. Providing fewer hidden nodes to the matrix representations prevented overfitting and allowed them to perform very

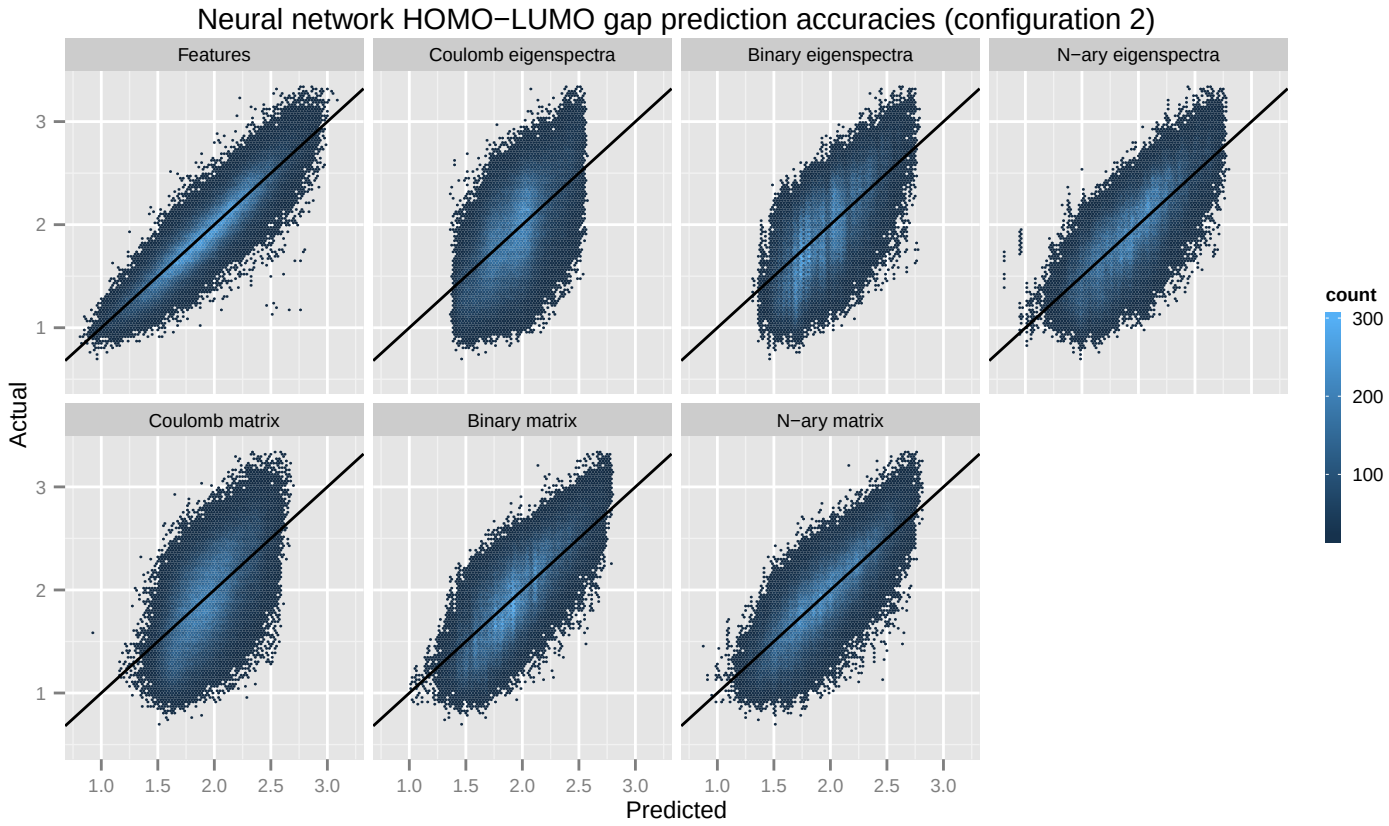


Figure 19: Scatter plot of actual value versus predicted value using on the test set using neural network configuration 2. The ideal actual = predicted line is overlaid.

well. Conversely, the eigenspectra representations performed worse in this configuration. This may be evidence for underfitting: we have taken away key degrees of flexibility that may have allowed the eigenspectra-based neural networks to learn patterns of the true model in the data. Nonetheless, the features representation still performed the best, which is consistent with the belief that this representation directly models the most important attributes of the input data.

Figure 19 illustrates the accuracy plots for this configuration. These plots somewhat resemble those in Figure 16, with the main difference being that the matrix representation predictions tend to have fewer fringe prediction points. Again, it is important to note that these networks do not appear to have converged. Additional training could result in substantial improvements to test set prediction.

Table 4: Comparison of results in this study

Representation	Best median OLS MSE	Best median ridge regression MSE	Best median LASSO regression MSE	Neural network MSE (config. 1)	Neural network MSE (config. 2)
Features	0.0943	0.0456	0.0562	0.0238	0.0258
Coulomb eigenspectra	0.1157	0.1157	0.1157	0.0986	0.1017
Binary eigenspectra	0.1054	0.1044	0.1051	0.0749	0.0860
n-ary eigenspectra	0.1199	0.1107	0.1108	0.0581	0.0655
Coulomb matrices				0.1304	0.1006
Binary matrices				0.1329	0.0643
n-ary matrices				0.1406	0.0494

4.5 Comparative analysis

Table 4 presents a summarization of the major results of this study. From these results, it is clear that the features representation across-the-board has the best performance. It also shows that, in general, the linear models perform fairly poorly compared to the neural networks. This suggests that the underlying HOMO-LUMO gap model has a nonlinear relationship with the predictors used in this study. However, with a poor neural network structure, we can severely overfit and actually perform worse than the simple linear models. Other than the features representation, the best MSEs in general were found using the matrix representations with the second neural network configuration. Within these, we found that the adjacency representations significantly outperformed the Coulomb representation.

We were only able to train two neural network configurations for this study due to limitations on computation time on the distributed cluster. However, these results are encouraging and lead us to believe that further research into these neural network methods may yield improved results. Different network structures and longer training times may lead to lower MSEs with relatively little required research time.

5 Conclusions and further research

This paper has devised and implemented a series of machine learning approaches to predict photovoltaic efficacy based on molecular covariates. This paper has examined the existing molecular representation of the Coulomb matrix. In addition, this paper

has also examined the features representation, which is underexamined in the literature, and proposed and examined the binary and n-ary adjacency representations. This paper has derived distributed linear and neural network regression algorithms that can scale to molecular databases of arbitrary size. Finally, this paper has implemented a scalable distributed system to perform molecular feature learning and analysis on the Harvard Clean Energy Project's database of 1.8 million organic molecules.

Overall, it appears that there is significant potential in determining photovoltaic efficacy of organic molecules using machine learning with almost no domain knowledge of chemistry or physics. The approaches used in this paper show that even simple linear models are able to achieve remarkably small test set mean squared errors. However, substantial gains can be achieved if the computational time can be afforded to train a neural network to perform molecular feature regression. This approach also has the advantage that it allows for regression over structured matrix data, which seems to lead to some of the best predictive models. The lowest test set MSEs were achieved using extracted features and adjacency matrix representations of molecules with neural networks.

The results observed in this paper are somewhat surprising given the previous literature. While this paper agrees with the previous literature that neural networks tend to be the superior learning approach, it disagrees in the best representation type. Most other studies examine Coulomb matrices and eigenspectra, citing that these representations are the most suitable for learning. However, in this study, we found that these representations made for relatively poor models. Our study suggests that extracted molecular features, a representation type that does not appear to be examined at depth in other studies, was the best representation type for regression. Furthermore, this paper found that the adjacency matrix representations had surprisingly good predictive power.

The research in this paper varies significantly from prior literature. While other studies

focus primarily on the learning approach, this paper also emphasizes the analysis of different representation types. Interestingly, the *overall* MSEs observed in this paper seem to be somewhat higher than in the literature. However, this is likely due to the differences in the datasets used. We found no similar study that performed regression over more than 10,000 molecules, compared to this study's 1.8 million. Additionally, the types of molecules examined vary significantly between studies. Whereas this study included molecules of up to 35 atoms, prior studies include molecules with a maximal size of 23 atoms per molecule [15].

The differences between this paper and the prior research demonstrates that there is still significant room for advancement in the area of molecular regression, due to the huge number of parameters in this space. Additional learning approaches and feature representations may offer promising improvements. Furthermore, no studies so far have exploited sophisticated domain-specific knowledge of quantum chemistry, which, combined with machine learning, may expand the type and efficacy of molecular representations. This is a burgeoning field, and continued research may help lead to the eventual development of a quantum machine.

6 References

[1] Joshua M Pearce. Photovoltaics - a path to sustainable futures. *Futures*, 34(7):663 – 674, 2002.

[2] Olivia Mah. Fundamentals of photovoltaic materials. pages 1–10, 1998.

[3] M Riede, T Mueller, W Tress, R Schueppel, and K Leo. Small-molecule solar cells—status and perspectives. *Nanotechnology*, 19(42):424001, 2008.

[4] Martin A. Green, Keith Emery, Yoshihiro Hishikawa, Wilhelm Warta, and Ewan D. Dunlop. Solar cell efficiency tables (version 42). *Progress in Photovoltaics: Research and Applications*, 21(5):827–837, 2013.

[5] A. Heeger. Low-cost plastic solar cells: a dream becoming a reality. In N. Stern V. Huber H. J. Schellnhuber, M. Molina and S. Kadner, editors, *Global Sustainability - A Nobel Cause*. Cambridge University Press, Cambridge, UK, 2010.

[6] Johannes Hachmann, Roberto Olivares-Amaya, Adrian Jinich, Anthony L. Appleton, Martin A. Blood-Forsythe, Laszlo R. Seress, Carolina Roman-Salgado, Kai Trepte, Sule Atahan-Evrenk, Suleyman Er, Supriya Shrestha, Rajib Mondal, Anatoliy Sokolov, Zhenan Bao, and Alan Aspuru-Guzik. Lead candidates for high-performance organic photovoltaics from high-throughput quantum chemistry - the harvard clean energy project. *Energy Environ. Sci.*, 2014.

[7] Gregoire Montavon, Matthias Rupp, Vivekanand Gobre, Alvaro Vazquez-Mayagoitia, Katja Hansen, Alexandre Tkatchenko, Klaus-Robert Mller, and O Anatole von Lilienfeld. Machine learning of molecular electronic properties in chemical compound space. *New Journal of Physics*, 15(9):095003, 2013.

[8] Charles Bergeron, Michael Krein, Gregory Moore, Curt M. Breneman, and Kristin P. Bennett. Modeling choices for virtual screening hit identification. *Molecular Informatics*, 30(9):765–777, 2011.

[9] ChemAxon Ltd. Chemaxon cheminformatics toolchain suite. `https://www.chemaxon.com/`, 1998–2013.

[10] Greg Landrum. Rdkit: A software suite for cheminformatics, computational chemistry, and predictive modeling. `http://www.rdkit.org/`, 2013.

[11] M. Rupp, A. Tkatchenko, K.-R. Müller, and O. A. von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Physical Review Letters*, 108:058301, 2012.

[12] Grgoire Montavon, Katja Hansen, Siamac Fazli, Matthias Rupp, Franziska Biegler, Andreas Ziehe, Alexandre Tkatchenko, Anatole von Lilienfeld, and Klaus-Robert Mller. Learning invariant representations of molecules for atomization energy prediction. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Lon Bottou, and Kilian Q. Weinberger, editors, *NIPS*, pages 449–457, 2012.

[13] Axel Grob Sonke Lorenz and Matthias Scheffler. Representating high-dimensional potential-energy surfaces for reactions at surfaces by neural networks. *Chemical Physical Letters*, 395, 2004.

[14] Sergei Manzhos and Tucker Carrington. A random-sampling high dimensional model representation neural network for building potential energy surfaces. *The Journal of Chemical Physics*, 125(8):–, 2006.

[15] Katja Hansen, Grégoire Montavon, Franziska Biegler, Siamac Fazli, Matthias Rupp, Matthias Scheffler, O. Anatole von Lilienfeld, Alexandre Tkatchenko, and Klaus-

Robert Müller. Assessment and validation of machine learning methods for predicting molecular atomization energies. *Journal of Chemical Theory and Computation*, 9(8):3404–3419, 2013.

[16] George Dahl, Alan McAvinney, and Tia Newhall. Parallelizing neural network training for cluster systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, PDCN '08, pages 220–225, Anaheim, CA, USA, 2008. ACTA Press.

[17] Lyle N. Long and Ankur Gupta. AIAA Paper No. 2005-7168 Scalable Massively Parallel Artificial Neural Networks.

[18] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-taylor, R.s. Zemel, P. Bartlett, F.c.n. Pereira, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. 2011.

[19] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In P. Bartlett, F.c.n. Pereira, C.j.c. Burges, L. Bottou, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1232–1240. 2012.

[20] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Bryan C. Catanzaro, and Andrew Y. Ng. Deep learning with cots hpc systems. In *ICML (3)*, volume 28 of *JMLR Proceedings*, pages 1337–1345. JMLR.org, 2013.

[21] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International*

Conference on Machine Learning, ICML '09, pages 873–880, New York, NY, USA, 2009. ACM.

[22] Rudi Helfenstein and Jonas Koko. Parallel preconditioned conjugate gradient algorithm on {GPU}. *Journal of Computational and Applied Mathematics*, 236(15):3584 – 3590, 2012. Proceedings of the Fifteenth International Congress on Computational and Applied Mathematics (ICCAM-2010), Leuven, Belgium, 5-9 July, 2010.

[23] Mingxian Xu, J.J. Miller, and E.J. Wegman. Parallelizing multiple linear regression for speed and redundancy: An empirical study. In *Distributed Memory Computing Conference, 1990., Proceedings of the Fifth*, pages 276–283, 1990.

[24] Gonzalo Mateos, Juan Andrs Bazerque, and Georgios B. Giannakis. Distributed sparse linear regression. *IEEE Transactions on Signal Processing*, 58(10):5262–5276, 2010.

[25] Albert P. Bartók, Mike C. Payne, Risi Kondor, and Gábor Csányi. Gaussian approximation potentials: The accuracy of quantum mechanics, without the electrons. *Phys. Rev. Lett.*, 104:136403, Apr 2010.

[26] Liefeng Bo and Cristian Sminchisescu. Greedy block coordinate descent for large scale gaussian process regression. *CoRR*, abs/1206.3238, 2012.

[27] Iain Murray. Gaussian processes and fast matrix-vector multiplies, 2009.

[28] Robert B. Gramacy, Jarad Niemi, and Robin Weiss. Massively parallel approximate gaussian process regression. *CoRR*, abs/1310.5182, 2013.

[29] Jie Chen, Nannan Cao, Kian Hsiang Low, Ruofei Ouyang, Colin Keng-Yan Tan, and Patrick Jaillet. Parallel gaussian process regression with low-rank covariance matrix approximations. *CoRR*, abs/1305.5826, 2013.

[30] Inc. Daylight Chemical Information Systems. Smiles - a simplified chemical language. http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html, Sept 2008.

[31] Jonathan D. Servaites, Mark A. Ratner, and Tobin J. Marks. Practical efficiency limits in organic photovoltaic cells: Functional dependence of fill factor and external quantum efficiency. *Applied Physics Letters*, 95(16):–, 2009.

[32] Harvard Research Computing Team. Odyssey: The research computing linux cluster. https://rc.fas.harvard.edu/kb/high-performance-computing/odyssey-the-research-computing-linux-cluster/, 2014.

[33] Harvard Research Computing Team. Advanced odyssey training. Slides hosted at https://software.rc.fas.harvard.edu/training/Advanced_Odyssey_Training.pdf, 2014.

[34] SchedMD. Slurm, the simple linux utility for resource management. http://slurm.schedmd.com/priority_multifactor.html, July 2012.

[35] Kevin P Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.

[36] L Eon Bottou and Yann Le Cun. Large scale online learning. In *In NIPS*, page 2004. MIT Press, 2003.

[37] Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In Léon Bottou, Olivier Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*. MIT Press, 2007.

[38] Warren S. Sarle. ai-faq/neural-nets/part3. ftp://ftp.sas.com/pub/neural/FAQ3.html, May 2001.